國立交通大學

資訊科學與工程研究所

碩 士 論 文

在資料串流環境探勘高實用性項目集之研究

Efficient Mining of High Utility Itemsets on Data Streams

研 究 生：黃心韻

指導教授：李素瑛　教授

中 華 民 國 九 十 六 年 七 月

在資料串流環境探勘高實用性項目集之研究

Efficient Mining of High Utility Itemsets on Data Streams

研 究 生：黃心韻　　Student：Hsin-Yun Huang

指導教授：李素瑛　　Advisor：Suh-Yin Lee

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

July 2007
Hsinchu, Taiwan, Republic of China

中華民國九十六年七月

在資料串流環境探勘高實用性項目集之研究

研究生：黃心韻　　指導教授：李素瑛

國立交通大學資訊科學與工程研究所

摘要

由於目前很多的應用如股市系統分析、線上交易等，資料都是以串流的形式產生，因此在資料串流環境中探勘有意義的樣式是一個很重要的課題。由於資料串流環境的限制，使探勘工作更爲複雜。探勘高實用性項目集是近年來新崛起的一個議題，依據使用者感興趣的主題，找尋出使用者所需要的樣式。在這樣的問題中，每個 item 的單位價格以及每筆交易中每個 item 出現的個數可以是任意值，因此更加深了問題的複雜度。

在這篇論文中我們提出了 *MHUI_TransSW* 以及 *MHUI_TimeSW*，有效率的在兩種滑動視窗的資料串流環境中探勘出高實用性的項目集。我們的方法，使用 TIDlist 或是位元向量去輔助紀錄 item 的資訊，再加上 lexicographical tree 的建立，改進了 THUI-Mine 演算法的效能。實驗結果也顯示出我們的方法，不管在時間還是空間上的使用，都能夠很有效率的在資料串流的環境中探勘出具有高實用性的項目集。

檢索詞：資料串流，滑動視窗，高實用性項目集，位元向量

Efficient Mining of High Utility Itemsets on Data Streams

Student: Hsin-Yun Huang Advisor: Suh-Yin Lee

Institute of Computer Science and Engineering
College of Computer Science
National Chiao-Tung University

Abstract

Since there are many applications in the form of data streams, such as sensor network, stock analysis, mining useful patterns from a data stream is an important issue nowadays. However, it is a difficult problem because of some limitations in the data stream environment. A new issue, called utility mining, for mining interesting pattern which is profitable for users is suggested in recent years. In the mining of high utility itemsets, the utility and the sales quantity of each item could be arbitrarily number, so many methods applied to frequent itemsets mining cannot be used anymore.

In this thesis, we propose *MHUI_TransSW* and *MHUI_TimeSW* to mine high utility itemsets on a data stream in two types of sliding window. We use *item information*, i.e. TIDlist or Bitvector of 1-itemsets, and lexicographical tree to improve the efficiency of THUI-Mine. The experiment results show that our approach efficiently find the high utility itemsets not only in execution time but also in memory space.

Index Terms: data stream, sliding window, high utility itemsets, Bitvector

Acknowledgement

I greatly appreciate the guidance from my advisor, Prof. Suh-Yin Lee. Without her suggestion and instruction, I can't complete this thesis.

Besides, I want to express my thanks to all the members in the Information System Laboratory and all my sincere friends for their suggestions and encouragements. Finally, I want to express my appreciation to my parents for their supports. This thesis is dedicated to them.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview and Motivation

Association rules mining (ARM) is one of the most widely used techniques in data mining and knowledge discovery and has tremendous applications in business, science and other domains. Standard methods for mining association rules are based on the support-confidence model. The first step involves finding all frequent itemsets, i.e., itemsets with support of at least *minsup*, and then, from these itemsets, generating all association rules with confidence of at least *minconf*. Once the frequent itemsets are found, generating association rules is straightforward and can be accomplished in linear time. Therefore, many researches focus on finding frequent itemsets efficiently.

Mining frequent itemsets has been widely studied over the last decade. Past research focuses on mining frequent itemsets from static database [1, 2, 3, 5, 6]. In many of the new applications, data flow through the internet or sensor network. It is challenging to extend the mining techniques to such a dynamic environment. The main challenges include a quick response to the continuous request, a compact summary of the data stream and a mechanism that adapts to the limited resources. An important research issue extended from the association rules mining is the discovery of temporal association patterns in data streams. However, most methods designed for the traditional databases cannot be directly applied to mining temporal patterns in data streams, since when transactions are added or expired, the support counts of the frequent itemsets contained in them are recomputed.

Traditional ARM model treats all the items in the database equally by only considering if an item is present in a transaction or not .However, the frequency of an itemset may not be a

sufficient indicator of interestingness, because it only reflects the number of transactions in the database that contain the itemset. It does not reveal the utility of an itemset, which can be measured in terms of cost, profit, or other expressions of user preferences. On the other hand, frequent itemsets may only contribute a small portion of the overall profit, whereas non-frequent itemsets may contribute a large portion of the profit.

Recently, to address the limitation of AMR, a utility mining model was defined [13]. Intuitively, utility is a measure of how "useful" (i.e. "profitable") an itemset is. The definition of utility of an itemset X**,** *u(X),* is the sum of the utilities of X in all the transactions containing X. The goal of utility mining is to identify high utililty itemsets which derive a large portion of the total utility. Traditional ARM model assumes that the utility of each item is 1 or 0, thus it is only a special case of utility mining, where the utility or the sales quantity of each item could be any number. If *u(X)* is greater than a utility threshold, X is a high utility itemset. Otherwise, it is a low utility itemset.

However, a high utility itemset may consist of some low utility items. A level-wise searching schema, Apriori, that exists in fast AMR algorithms, is used to prune impossible itemsets as soon as possible. However this property cannot apply to the utility mining model. Without this property, the number of candidates generated at each level quickly approaches all the combinations of all the items. We are confronted by two difficulties. The first is how to restrict the size of the candidate set and simplify the computation for calculating the utility. The second is how to find temporal high utility itemsets from data streams as time advances.

In this thesis, we propose a method that can find high utility itemsets from data streams efficiently and effectively. We use the downward closure property in Two-Phase Algorithm [19], and add an efficient method to restrict the candidates generated and simplify the computation of utility.

1.2 Related Work

The problem of generating association rules was first introduced in [1] and an algorithm called *AIS* was proposed for mining all association rules. In past ten years, a considerable number of studies have been made on traditional ARM algorithms and optimizations. The base of these traditional ARM algorithms is the "downward closure property" (anti-monotone property): any subset of a frequent itemset must also be frequent. That is, only the frequent k-itemsets are exploited to generate potential frequent (k+1)-itemsets, called candidates. This kind of candidate-generate-and-test methods needs multiple scans of database. A subsequent research is proposed to speed-up Apriori, such as DHP [5] which uses a hash function to prune candidate 2-itemsets. Besides, there are also other investigations for finding frequent itemsets. Partition [3] algorithm is a kind of filter-and-refine approach. It first generates candidate itemsets and then in one more scan verifies the validity of each candidate itemset. Thus, the method needs only two scans of database. FP-growth [6] algorithm is a pattern-growth approach. It completely eliminates the candidate generation bottleneck by using a new tree structure called Frequent Pattern Tree (FP-Tree) which is constructed in only two scans, and then recursively mines FP-trees of decreasing size to generate large itemsets without candidates generation and database scans.

One of the key features of all the previous algorithms is that they just suited for static databases. However, most methods designed for the traditional databases cannot be directly applied for mining temporal patterns in data streams because of high complexity. In recent years, processing data from data streams is a very popular topic in data mining. Three models are adopted by many researchers in ways of time spanning [8]: landmark model, sliding window model, and damped window model. Landmark model utilizes all the data between a particular point of time (called landmark) and the current time for mining. Lossy-counting [9] is the representative approach under the landmark model. Li et al [15] proposes DSM-FI

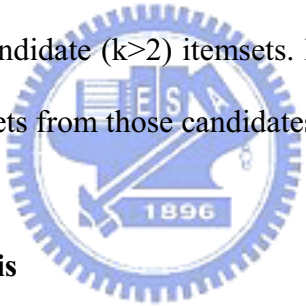
algorithm, which is a projection-based and single passed algorithm, to mine frequent itemsets in the landmark model over a data stream. However, in many applications, new data are often more important than old ones. The landmark model is not aware of time and therefore cannot distinguish between new data and old ones. Therefore, the time-fading model, a variation of the landmark model, has been presented. It assigns different weights to transactions such that the new ones have higher weights than old ones. EstDec algorithm [13] uses a decay function to reduce the weight of the old transactions. In some applications, users can only be interested in the data recently arriving within a fixed time period, thus the sliding window model proposed. Algorithms SWF [7] employs a filtering threshold in each partition to deal with the candidate itemsets generation. Algorithm Moment [16] use the closed enumeration tree (CET), to maintain a dynamically selected set of itemsets over a sliding window.

A formal definition of utility mining and theoretical model was proposed in [17], namely MEU, where the utility is defined as the combination of utility information in each transaction and additional resources. Since this model cannot rely on downward closure property of Apriori to shrink the number of candidate itemsets, a heuristic approach is used to predict whether an itemset should be added to the candidate set. However, this prediction usually overestimates, especially at the beginning stages, where the number of candidates approaches the number of all the combinations of items. The examination of candidates is impractical, either in computation cost or in memory space cost whenever the number of items is large or the utility threshold is low. Besides, this model may miss some high utility itemsets when the variation of the itemsets supports is large.

Another algorithm named Two-Phase [19], which is based on the definition in [17] achieves finding high utility itemsets. It presented a Two-Phase algorithm that not only can prune down the number of candidate itemsets, but also find the complete high utility itemsets. In first phase, it defines a *transaction-weighted utilization mining model* that holds a "*Transaction-Weighted Downward Closure Property*". The size of candidate set is reduced by only

considering the supersets of high transaction-weighted utilization itemsets. In second phase, only one extra database scan is performed to filter out the high transaction-weighted utilization itemsets that are indeed low utility itemsets. This algorithm guarantees that the complete set of high utility itemsets will be defined. However, Two-Phase algorithm is focused just only on traditional databases and is not suited for data streams.

Chu et al [20] propose THUI (Temporal High Utility Itemsets)-Mine algorithm, which is the first work on mining temporal high utility itemsets from data streams. The underlying idea of THUI-Mine algorithm is to integrate the advantages of Two-Phase algorithm [19] and SWF [7] algorithm and augment with the incremental mining techniques for mining temporal high utility itemsets. In the first scan of database, it employs a filtering threshold in each partition to generate progressive transaction-weighted utilization set of itemsets, and then uses database scan reduction to generate k-candidate (k>2) itemsets. Finally, It just needs one more scan to find temporal high utility itemsets from those candidates.



1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. Some basic definitions and terminology about utility itemsets, and sliding window are described in Chapter 2. Our proposed method for mining high utility itemsets is presented in Chapter 3. The experiments and performances are described in Chapter 4. Conclusion and future work is in Chapter 5.

Chapter 2

Problem Definition and Background

In this chapter we introduce the basic definition of problems. We introduce the data stream environment and the sliding window model in Section 2.1. Next we describe the definition of utility itemsets and the problem of mining high utility itemsets in Section 2.2. Finally, we describe transaction-weighted utilization closure property in Section 2.3.

2.1 Definition and Background of Data Stream

2.1.1 Data Stream

Database and knowledge discovery communities have focused on a new data model, where data arrive in the form of continuous streams. It is often referred to as *data streams* or *streaming data*. The characteristics of data streams are as follows: (1) Continuity: data continuously arrive at a rapid rate. (2) Expiration: Data can be read only once. (3) Infinity: The total amount of data is unbounded.

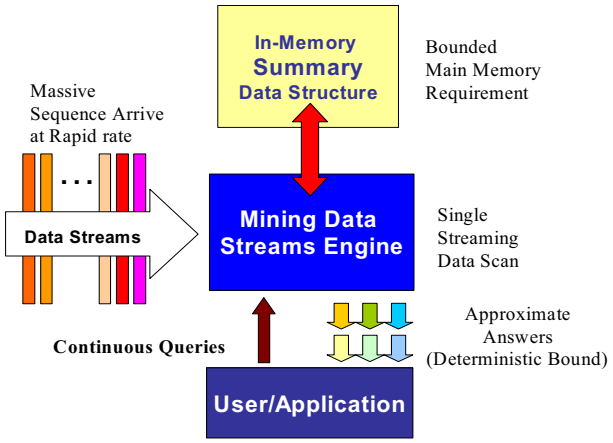
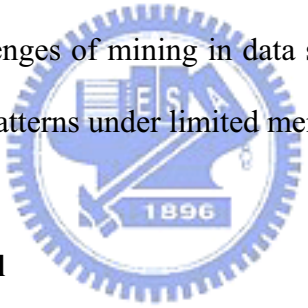


Figure 2-1. Data stream environment

Figure 2-1 shows the data stream environment. For reasons given above, mining patterns of data streams differs from traditional mining of static database in the following aspects: Firstly, data streams continuously arrive at a rapid rate and thus the amount of data is huge. This means that once a new data element arrives, it must be processed quickly. Besides, once a data element is removed from the main memory, it is unable to backtrack over previously-arrived data elements. Therefore, the best condition is to achieve one sequential pass over the data, called one-pass scan. Secondly, the relatively small memory compared with the large amount of streaming data results in the fact that we can only store a concise summary or partial data of the data stream. Finally, due to the limited memory and one-pass scan, getting precise answers from data streams is commonly impossible or very difficult. Due to these reasons it is not feasible to use traditional multiple-pass techniques for mining static databases in the data stream environment. The challenges of mining in data streams are how to design an efficient algorithm to derive the useful patterns under limited memory and execution time.



2.1.2 A Sliding Window Model

Some applications in data streams emphasize the importance of the recent transactions. A sliding window model is suitable to solve this kind of problems. In the sliding window model, knowledge discovery is performed over a fixed number, window size, of recently generated data elements which is the target of data mining and once the window is full, ***window sliding*** is performed to eliminate the oldest data and then append the newest data.

According to the basic unit of window sliding, two types of sliding window, i.e., **transaction-sensitive sliding window** *(TransSW)* and **time-sensitive sliding window** (*TimeSW*) are used in mining data streams. A transaction-sensitive sliding window in the transaction data stream is a window that slides forward for *every transaction*, whereas a time-sensitive sliding window in the transaction data stream is a window that slides forward for

e*very time unit*(TU_i), each consisting of variable number of transactions. Therefore, the window size, *w*, in *TransSW* at each slide is a fixed number of transactions, whereas the window size, *w*, in *TimeSW* at each slide is a variable number of transactions. The sliding window model is shown in Figure 2-2.

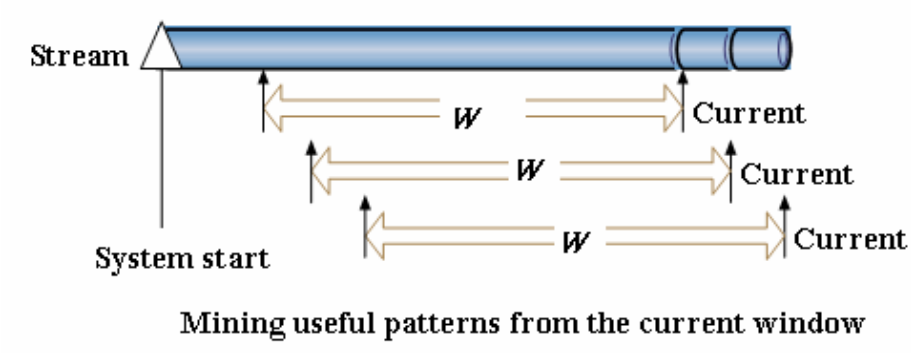


Figure 2-2. The sliding window model

2.2 Problem Definition: Mining High Utility Itemsets in a Sliding Window Model

2.2.1 Utility Itemsets

Let $I = \{i_1, i_2, ..i_n\}$ be a set of *n* distinct literals called *items*. $D = \{T_1, T_2, ..., T_m\}$ is a set of variable length transactions where each transaction $T_i \in D$ is a subset of *I*. A transaction also has an associated unique identifier called TID. In general, a set of items is called an itemset. The number of items in an itemset is called the *length* of an itemset. Itemsets of length *k* are referred to as *k-itemsets*.

In traditional frequent itemsets mining, the number of an item in each transaction is always 0 or 1. However, in utility mining model, the number of an item in each transaction, called **local transaction utility**, may be arbitrary number. An extra resource, called **external utility** which can be a measure for describing user preference, is defined as a **utility table**. Figure 2-3 shows an example of the transaction database and a utility table.

item / TID	a	b	c	d	e
T1	0	0	26	0	1
T2	0	6	0	1	0
T3	12	0	0	1	0
T4	0	1	0	7	0
T5	0	0	12	0	2
T6	1	4	0	0	1
T7	0	10	0	0	1
T8	1	1	1	3	1
T9	2	1	27	0	2
T10	0	6	2	0	0
T11	0	3	0	2	0
T12	0	2	1	0	0

(a) Transaction table

item	Profit (Per Unit)
a	3
b	10
c	1
d	6
e	5

(b) Utility table

Figure 2-3. An example of input transaction database and utility table

Some definitions of a set of items that leads to the formal definition of utility mining problem is given in [17]

1. $o(i_p, T_q)$, ***local transaction utility***, represents the quantity of item i_p in the transaction T_q. For example, $o(a, T_3) = 12$ and $o(c, T_5) = 12$.
2. $u(i_p)$, ***external utility***, is the value associated with item i_p in the utility table. For example, u(a)=3 and u(b)=10.
3. $u(i_p, T_q)$, utility of an item i_p in T_q, is defined as $o(i_p, T_q) \times u(i_p)$ For example, $u(b, T_2) = o(b, T_2) \times u(b) = 6*10=60$ and $u(d, T_2) = o(d, T_2) \times u(d) = 1*6=6$.
4. $u(X, T_q)$, utility of an itemset X in transaction T_q, is defined as $\sum_{i_p \in X} u(i_p, T_q)$, where $X = \{i_1, i_2, ..., i_k\}$ is a k-itemset, $X \subseteq T_q$ and $1 \le k \le n$. For example, $u(bd, T_2) = u(b, T_2) + u(d, T_2) = 6*10+1*6=66$, $u(bd, T_4) = u(b, T_4) + u(d, T_4) = 1*10+7*6=52$, $u(bd, T_8) = u(b, T_8) + u(d, T_8) = 1*10+3*6=28$, $u(bd, T_{11}) = u(b, T_{11}) + u(d, T_{11}) = 3*10+2*6=42$.
5. $u(X)$, **utility of an itemset *X***, is the sum of the utilities of X in all the transactions

containing X, is defined as $\sum_{T_q \in D \wedge X \subseteq T_q} u(X, T_q)$. For example, $u(bd) = u(bd, T_2) + u(bd, T_4) + u(bd, T_8) + u(bd, T_{11}) = 66+52+28+42 = 188$. The goal of utility mining is to identify high utility itemsets which derive a large portion of the total utility. If the twelve transactions are the target of data mining and the minimum utility threshold is 120, bd is a high utility itemset.

2.2.2 Problem Definition: Mining High Utility Itemsets in a Sliding Window Model

A transaction-sensitive sliding window ($TransSW$) in the transaction data stream is a window that slides forward for every transaction. The window at each slide has fixed number, w, of transactions, and w is called the size of the window. The current transaction-sensitive sliding window is $TransSW_{N-w+1} = [T_{N-w+1}, T_{N-w+2}, ..., T_N]$, where *N-w+1* is the id of current window. An itemset X is called a high utility itemset if $u(X) \geq ut \times w$, where ut is a user specified minimum utility threshold in the range of [0,1]. The value $ut \times w$ is the minimum utility in the current transaction-sensitive sliding window.

A time-sensitive sliding window ($TimeSW$) in the transaction data stream is a window that slides forward for every time unit. Each time unit TU_i consists of a variable number, $|TU_i|$, of transactions, and $|TU_i|$ is also called the size of the time unit. Due to the different size of each time unit, the window at each slide has variable number of transactions. The current time-sensitive sliding window is $TimeSW_{N-w+1} = [TU_{N-w+1}, TU_{N-w+2}, ..., TU_N]$, where *N-w+1* is the id of current window. An itemset X is called a high utility itemset if $u(X) \geq ut \times |TimeSW_{N-w+1}|$, where ut is a user specified minimum utility threshold in the range of [0, 1] and $|TimeSW_{N-w+1}| = |TU_{N-w+1}| + |TU_{N-w+2}| + ... + |TU_N|$ is the number of transactions in the current time-sensitive sliding window, called window size. The value

$ut \times |TimeSW_{N-w+1}|$ is the minimum utility in the current time-sensitive sliding window.

Later in thesis, we will show that our method can be adopted in both of transaction-sensitive and time-sensitive sliding window model.

2.3 Transaction-Weighted Downward Closure Property

The downward closure property of Apriori cannot be applied for the utility mining model. For example, *u(d)*=14*6=84<120 and is a low utility itemset but its superset *u(bd)*=160>120 is a high utility itemset. If candidates generated use all the combinations of items, the computation will be intolerable. A level-wise approach apply for utility mining, called "*Transaction-weighted Downward Closure Property* " is proposed in Two-Phase Algorithm [19].

Definition 1. (Transaction Utility) The transaction utility of transaction T_q, denoted as $tu(T_q)$, is the sum of the utilities of all items in T_q. For example, $tu(T_2) = u(b,T_2) + u(d,T_2)$=6*10+1*6=66

Definition 2. (Transaction-Weighted Utilization) The transaction-weighted utilization of an itemset *X*, denoted as ***twu(X)***, is the sum of the transaction utilities of all the transactions containing *X*. Assume the target of data mining is T1 to T9, $twu(bd) = tu(T_2) + tu(T_4) + tu(T_8)$= 66+52+37=155

Definition 3. (High Transaction-Weighted Utilization Itemsets) *X* is a high transaction-weighted utilization itemset if $twu(X) \geq$ minimum utility. Assume the minimum utility is 120, and thus bd is a high transaction-weighted utilization itemset.

Theorem 1. (Transaction-Weighted Downward Closure Property) Let I^K be a k-itemset and I^{K-1} be a (k-1)-itemset such that $I^{K-1} \subset I^K$. If I^K is a high transaction-weighted utilization itemset, I^{K-1} is a high transaction-weighted utilization itemset.

Chapter 3

An Efficient Mining of High Utility Itemsets

The goal of our work is to find an efficient method for mining high utility itemsets in a data stream. Therefore, in Section 3.1 we introduce a related work, called THUI-Mine algorithm. Next, we introduce our proposed method for mining high utility itemsets in a transaction-sensitive sliding window model, denoted as MHUI_TransSW, in Section 3.2. Subsequently, we extend this method to time-sensitive sliding window model, denoted as MHUI_TimeSW, in Section 3.3.

3.1 Related Work: THUI (Temporal High Utility Itemsets)-Mine Algorithm

THUI-Mine [20] is based on transaction-weighted downward closure property, and is extended the property with the sliding-window-filtering technique to find the temporal high utility itemsets over a sliding window. In essence, by partitioning a transaction database into several partitions from data streams, algorithm THUI-Mine employs a filtering threshold in each partition to deal with the transaction-weighted utilization itemsets generation.

For ease of exposition, the processing of a partition is termed a phase of processing. The cumulative information in the prior phase is selectively carried over toward the generation of candidate itemsets in the subsequent phases. The cumulative information THUI-Mine maintained consists of these two summary structures:

1. progressive transaction-weighted utilization set of itemsets (also called potential candidate 2-itemsets): composed of the following two types of itemsets, i.e.,

   (1) The transaction-weighted utilization itemsets that were carried over from the

previous progressive candidate set in the previous phase and remain as transaction-weighted utilization itemsets after the current partition is taken into consideration.

(2) The transaction-weighted utilization itemsets that were not in the progressive candidate set in the previous phase but are newly selected after the current partition is taken into consideration.

2. $TUP_k(I)$: The transaction-weighted utilization itemsets and its corresponding transaction -weighted utility in each partition P_K.

After processing a partition P_K, THUI-Mine maintains the potential candidate 2-itemsets and $TUP_k(I)$. Each potential candidate 2-itemset $c \in C_2$ has two attributes: (1) ***c.start*** contains the identify the starting partition identifier when c was added to C_2, and (2) ***twu(c),*** transaction-weighted utility of itemset *c*, is the sum of the transaction utilities of all the transactions containing c since c was added to C_2. Table 3-1 shows the meanings of symbols used in THUI-Mine. The mining process of THUI-Mine is decomposed into two processes:

1. ***The preprocessing procedure:*** While the window is not full yet, it deals with mining on the original transaction database, e.g., $db^{1,n}$. This procedure is described in Section 3.1.1.
2. ***The incremental procedure:*** While the window is full and new partition arrives, it needs to slide the window. Thus the cumulative information needs to be updated. This procedure is described in Section 3.1.2.

Table 3-1. The meanings of symbols used in THUI-Mine

$db^{i,j}$	Partition database from P_i to P_j
s	Utility threshold in one partition
$TUP_k(I)$	Transactions in P_k that contain itemset *I* with transaction utility
$Thtw^{i,j}$	The progressive temporal high transaction-weighted utilization 2-itemsets of $db^{i,j}$

Δ^-	The deleted portion of an ongoing database
D^-	The unchanged portion of an ongoing database
Δ^+	The added portion of an ongoing database

3.1.1 The Preprocessing Procedure

Figure 3-1 shows an input transaction database and utility table. Let each partition contains three transactions and each window contains nine transactions. Assume the minimum utility is 120 for nine transactions, and thus the filtering threshold is s=120/3=40 for each partition. The first window, $db^{1,3}$, is segmented into three partitions, i.e., $\{P_1, P_2, P_3\}$. Each partition is scanned sequentially for the generation of **progressive temporal high transaction-weighted utilization 2-itemsets of** $db^{1,3}$, $Thtw^{1,3}$, in the first scan. Figure 3-2 shows the transaction utility of each transaction.

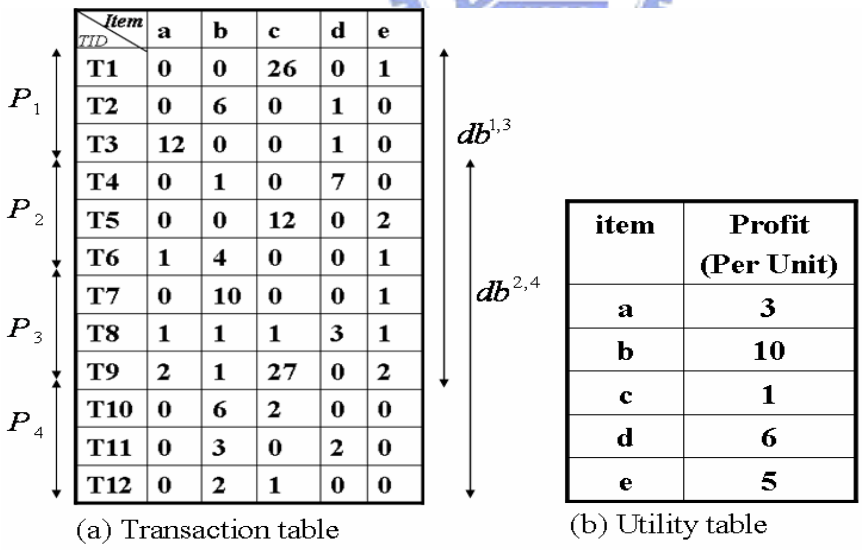


Figure 3-1. An example of input transaction database and utility table

TID	Transaction utility	TID	Transaction utility
T1	31	T7	105
T2	66	T8	37
T3	42	T9	53
T4	52	T10	62
T5	22	T11	42
T6	48	T12	21

Figure 3-2. The transaction utility of each transaction

After scanning the first 3 transactions, i.e., partition P_1, we use high transaction-weighted utilization 1-itemsets, itemsets {a, b, d}, in P_1 to generate potential candidate 2-itemsets {ab, ad, bd}. Itemsets {ab, ad, bd} are newly generated in partition P_1, so the start value of them is 1, the identifier of partition P_1. The transaction-weighted value of itemsets {ab, ad, bd} in P_1 are 0, 42, 66 respectively. Since *twu(ab)*=0<40, itemset ab is removed. On the contrary, itemsets {ad, bd}, in shaded portion, have transaction-weighted value greater than 40, so they are the potential candidate 2-itemsets remained and then its information are carried over to the next phase of processing. $TUP_1(I)$ maintains these potential candidate 2-itemsets and its transaction -weighted utility in partition P_1. Figure 3-3 shows the potential candidate 2-itemsets and $TUP_1(I)$ after processing P_1.

$db^{1,1}(P_1)$		
C_2	start	twu
ab	1	0
ad	1	42
bd	1	66

$TUP_1(I)$	
C_2	twu
ad	42
bd	66

Figure 3-3.The potential candidate 2-itemsets and $TUP_1(I)$ after processing P_1

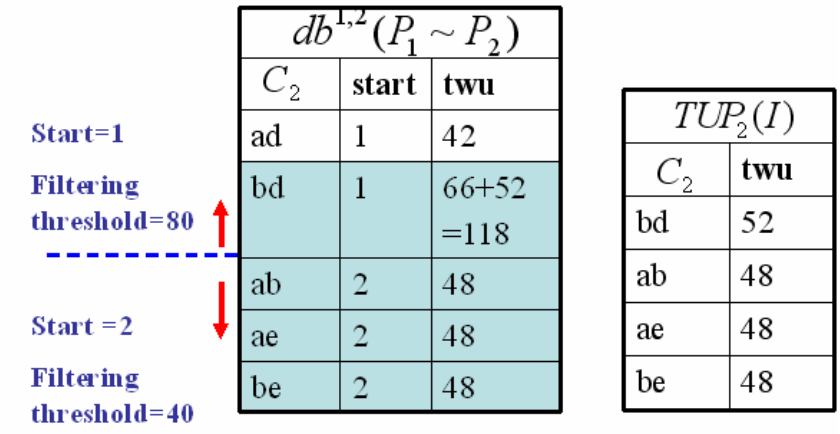
After processing partition P_2, candidate 2-itemsets are decomposed into two types:

(1) Itemsets that are carried over from the previous phase P_1. The start value of this kind of itemsets is 1. For example, itemsets {ad, bd} are carried over from P_1, so *ad.start*=1 and *bd.start*=1. Though itemset bd is carried from P_1, it also occurs in P_2. The transaction-weighted utility of bd in P_2 is maintained in $TUP_2(I)$. Besides, its transaction-weighted utility is accumulated and *twu(bd)* becomes 66+52=118.

(2) Itemsets that are newly identified after the current partition, P_2, is taken into consideration.

The start value of this kind of itemsets is 2. For example, itemsets {ab, ae, be} are newly generated after P_2 is taken into consideration.

THUI-Mine prunes those itemsets with different filtering threshold. For itemset c where *c.satrt*=1, its filtering threshold is 2*s=40*2=80. For itemset c where *c.start*=2, its filtering threshold is 1*s=40. For example, *Twu(ad)*=42<80, so itemsets ad is not carried to next partition. *Twu(bd)*=118>80 and *twu(ab)*=48>40, so itemset bd and ab are carried over to the next partition. After pruning, there are four potential candidate 2-itemsets {ab, ae, bd, be}, in shaded portion, carried over to the next phase. One of them is carried over from partition P_1, and three of them are newly identified in P_2. Figure 3-4 shows the potential candidate 2-itemsets and $TUP_2(I)$ after processing P_2.



$db^{1,2}(P_1 \sim P_2)$

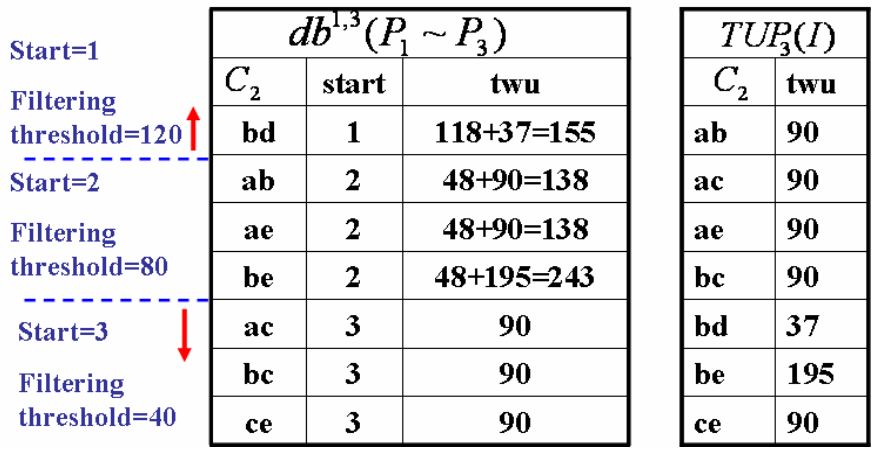
C_2	start	twu
ad	1	42
bd	1	66+52 =118
ab	2	48
ae	2	48
be	2	48

$TUP_2(I)$

C_2	twu
bd	52
ab	48
ae	48
be	48

Figure 3-4.The potential candidate 2-itemsets and $TUP_2(I)$ after processing P_2

Partition P_3 is processed in the same way. Figure 3-5 shows the potential candidate 2-itemsets and $TUP_3(I)$ after processing P_3. Observe that there are seven potential candidate 2-itemsets left in the preprocessing procedure.



$db^{1,3}(P_1 \sim P_3)$

C_2	start	twu
bd	1	118+37=155
ab	2	48+90=138
ae	2	48+90=138
be	2	48+195=243
ac	3	90
bc	3	90
ce	3	90

$TUP_3(I)$

C_2	twu
ab	90
ac	90
ae	90
bc	90
bd	37
be	195
ce	90

Figure 3-5. The potential candidate 2-itemsets and $TUP_3(I)$ after processing P_3

After generating $Thtw^{1,3}$, THUI-Mine employs the scan reduction technique to generate other candidate itemsets. It uses $Thtw^{1,3}$ to generate candidate 3-itemsets, C_3, and subsequently use candidate *(k-1)*-itemsets, C_{k-1}, to generate candidate *k*-itemsets, C_k (k= 3,…n), where C_n is the candidate last-itemsets. For instance, abe is constructed because itemsets {ab, ae, be} is in $Thtw^{1,3}$. Candidate 3-itemsets {abe, abc, bce} is the last candidate itemsets in this example. After generating all candidate itemsets, one more scan is needed to find the high utility itemsets. Table 3-2 shows the itemsets generated after first and second scan of $db^{1,3}$.

Table 3-2. The itemsets generated after first and second scan of $db^{1,3}$

Candidate itemsets (1^{st} scan of $db^{1,3}$)	High utility itemsets (2^{nd} scan of $db^{1,3}$)
a, b, c, d, e ab, ac, ae, bc, bd, be, ce abc, abe, ace, bce abce	b, bd, be

3.1.2 The Incremental Procedure

When partition P_4 arrives, window sliding is performed. As depicted in Figure 3-1, the current window will be moved from $db^{1,3}$ to $db^{2,4}$, i.e., $\{P_2, P_3, P_4\}$. Some transactions, i.e., T1, T2 and T3, are deleted from the window, and transactions T10, T11 and T12 are added. This incremental procedure is decomposed into three sub-steps as follows:

1. **Prune the oldest partition and update potential candidate 2-itemsets**

   In this sub-step, we check the pruned partition, P_1, and reduce the value of transaction-weighted utility and set *c.start=2* for those potential candidate 2-itemsets where *c.start*=1. For example, Figure 3-5 shows the potential candidate 2-itemsets after processing $db^{1,3}$. Observe that *bd.start*=1, i.e. bd is in the pruned partition P_1. We can observe that *twu(bd)*=66 in P_1 from $TUP_1(I)$. Hence after pruning P_1, *twu(bd)*=155-66=89 and *bd.start* = 2. Figure 3-6 shows the result after the first sub-step, where $D^- = db^{1,3} - P_1$.

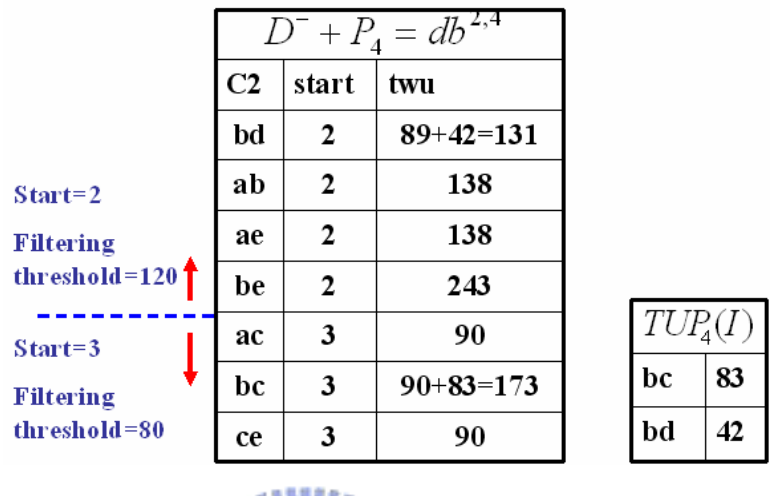
$D^- = db^{1,3} - P_1$		
C2	**start**	**twu**
bd	**1** 2	**155-66=89**
ab	**2**	**138**
ae	**2**	**138**
be	**2**	**243**
ac	**3**	**90**
bc	**3**	**90**
ce	**3**	**90**

Figure 3-6. The potential candidate 2-itemsets after performing first sub-step

2. **Append newest partition and update potential candidate 2-itemsets**

   In this sub-step, the process to add new partition, P_4, is similar to the operation of partition P_2, in the preprocessing process. There is no new itemsets join the potential candidate 2-itemsets. However, itemsets {bc, bd} are carried from previous phase and also

appears in P_4, so their transaction-weighted utility are accumulated and maintain them to $TUP_4(I)$. Figure 3-7 shows the potential candidate 2-itemsets and $TUP_4(I)$ after processing P_4, where $D^- + P_4 = db^{2,4}$.



$D^- + P_4 = db^{2,4}$

C2	start	twu
bd	2	89+42=131
ab	2	138
ae	2	138
be	2	243
ac	3	90
bc	3	90+83=173
ce	3	90

$TUP_4(I)$	
bc	83
bd	42

Figure 3-7. The potential candidate 2-itemsets and $TUP_4(I)$ after performing second sub-step

3. Use *scan reduction techniques* to generate all candidate itemsets, as mentioned above, and then one more database scan finds the temporal high utility itemsets of $db^{2,4}$. Table 3-3 shows the itemsets generated after first and second scan of $db^{2,4}$.

Table 3-3. The itemsets generated after first and second scan of $db^{2,4}$

Candidate itemsets (1^{st} scan of $db^{2,4}$)	High utility itemsets (2^{nd} scan of $db^{2,4}$)
a,b,c,d,e ab, ac, ae, bc, bd, be, ce abc, abe, ace, bce abce	b, bd, be

3.1.3 The Drawback of THUI-Mine Algorithm

THUI-Mine may give rise to two possible problems as follows:

1. More false candidate itemsets:

The temporal high transaction-weighted utilization itemsets may contain itemsets which may not be truly high utility itemsets, called **false candidates**. The number of false candidates depends on many factors such as the characteristics of the data, how the data is partitioned, number of partitions, and so on. THUI-Mine records the start partition of each potential candidate 2-itemsets and then uses different filtering threshold to prune. In this way, THUI-Mine may overestimate some itemsets concentrating in the later partitions. For example, there are seven potential candidate 2-itemsets maintained in $db^{1,3}$ shown in Figure 3-5. Observe that *ac.start*=3 and *bc.start*=3. Since *twu(ac)=twu(bc)*>40, filtering threshold, itemsets ac and bc are added when P_3 is taken into consideration. In fact, itemsets ac and bc only occur in P_3. In other words, itemets ac and bc are overestimated.

Next, THUI-Mine uses *scan reduction technique* to generate candidate itemsets. Candidate 3-itemsets, C_3, is generated from $Thtw^{i,j} \times Thtw^{i,j}$. Subsequently, C_4 is generated from $C_3 \times C_3$, where C_3 will have a size greater than high transaction-weighted utilization 3-itemsets. In other words, once the number of $Thtw^{i,j}$ increases, it leads to a chain reaction for C_K (k=3,…,n). Later in experiments, we will show that if the size of a partition or the minimum utility decreases, the situation will be getting worse.

2. More memory:

THUI-Mine needs to maintain $TUP_k(I)$ for use in the incremental procedure. The memory varies with the number of candidate 2-itemsets affected by many factors mentioned above. Since THUI-Mine has these disadvantages, we propose a new method in next Section so as to reduce the number of candidate itemsets generated, and decrease the memory used. Later in Chapter 4, experiments will show that the proposed method outperforms than THUI-Mine algorithm.

3.2 Our Proposed Method: MHUI_TransSW

In this section, we propose an efficient method, called **MHUI_TransSW**(Mining High Utility Itemsets over a Transaction-sensitive Sliding Window)**,** to mine the set of all high utility itemsets with a transaction-sensitive sliding window. MHUI_TransSW is based on *transaction-weighted downward closure property* and additionally use effective *item information*, i.e., TIDlist or Bitvector of all 1-itemsets, to restrict candidate itemsets generated and thus reduce the time and memory needed. In section 3.2.1 we describe the representation of *item information* and then proposed an efficient method, called MHUI_TransSW, in Section 3.2.2.

3.2.1 Representation of *Item information* (TIDlist or Bitvector of items)

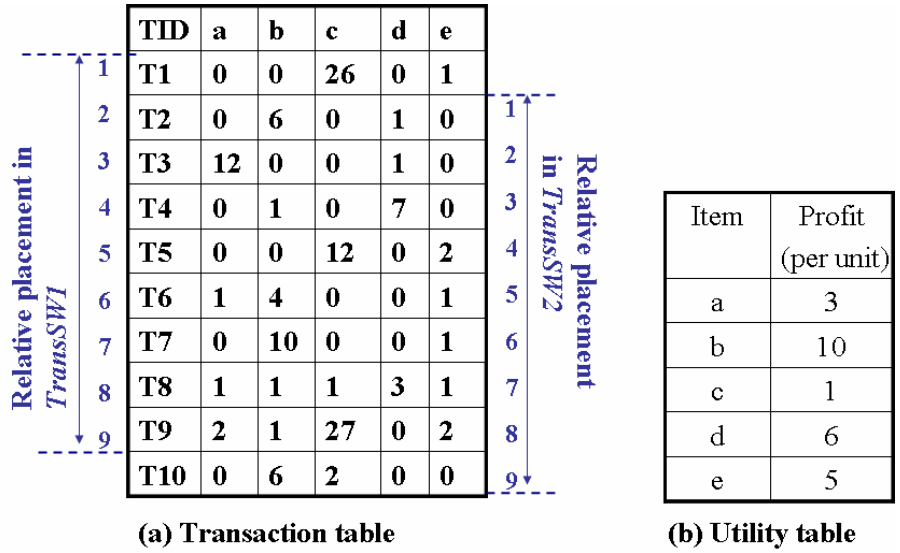
For each item x, *item information*, i.e. TIDlist(x) or Bitvector(x), maintains the relative placement of all transactions containing x in each sliding window, so that we can reduce the scan of transaction database. Assume the window contains *w* transactions, i.e. window size is *w*. The representation of *item information* is described as follows.

1. **Definition of Bitvector(x):** For each item x in the current transaction-sensitive sliding window *TransSW*, a bit-sequence with *w* bits, denoted as Bitvector(x)**,** is constructed. If an item x is in the *i*-th transaction of current *TransSW*, the *i*-th bit of Bitvector(x) is set to be 1; otherwise, it is set to be 0.
2. **Definition of TIDlist(x):** For each item x in the current transaction-sensitive sliding window *TransSW*, a sorted list with at least *w* value, denoted as TIDlist(x), is constructed. If an item x is in the *i*-th transaction of current *TransSW*, TIDlist(x) contains *i*.

Figure 3-8 shows an example of utility table and input transaction database where the first two sliding windows are marked by $TransSW_1$ and $TransSW_2$. Assume that the window size

is 9, $TransSW_1$ consists of T1 to T9, and $TransSW_2$ consists of T2 to T10.

As reading the current transaction-sensitive sliding window, the *item information* generates. The TIDlist and Bitvector of all items in each window are listed in Table 3-4. Take item a as example. a appears in T3, T6, T8, and T9 which is in the 3rd, 6th, 8th, 9th placement in $TransSW_1$ respectively. Therefore, Bitvector(a)=<001001011> and TIDlist(a)={3,6,8,9} in $TransSW_1$. a also appears in T3, T6, T8, and T9, however, the relative placement is in the 2rd, 5th, 7th, 8th in $TransSW_2$ respectively. Bitvector(a)=<010010110> and TIDlist(a)={2,5,7,8} in $TransSW_2$.



TID	a	b	c	d	e
T1	0	0	26	0	1
T2	0	6	0	1	0
T3	12	0	0	1	0
T4	0	1	0	7	0
T5	0	0	12	0	2
T6	1	4	0	0	1
T7	0	10	0	0	1
T8	1	1	1	3	1
T9	2	1	27	0	2
T10	0	6	2	0	0

(a) Transaction table

Item	Profit (per unit)
a	3
b	10
c	1
d	6
e	5

(b) Utility table

Figure 3-8. An example of transaction database and utility table

Table 3-4. The TIDlist and Bitvector of all items in the first two windows

Item \ TIDlist(x)	$TransSW_1$	$TransSW_2$
a	{3,6,8,9}	{2,5,7,8}
b	{2,4,6,7,8,9}	{1,3,5,6,7,8,9}
c	{1,5,8,9}	{4,7,8,9}
d	{2,3,4,8}	{1,2,3,7}
e	{1,5,6,7,8,9}	{4,5,6,7,8}

Item \ Bitvector(x)	$TransSW_1$	$TransSW_2$
a	<001001011>	<010010110>
b	<010101111>	<101011111>
c	<100010011>	<000100111>
d	<011100010>	<111000100>
e	<100011111>	<000111110>

3.2.2 MHUI_TransSW Method

We adopt the sliding window model to mine the high utility itemsets in a transaction-sensitive sliding window. The mining process consists of three phases, i.e., window initialization phase, window sliding phase, and high utility itemsets generation phase, described from Section 3.2.2.1 to 3.2.2.3, respectively. Table 3-5 shows the meanings of symbols used in our work. We assume only the summary structure derived from previous window is provided for mining high utility itemsets in current window. The summary structure of MHUI_TransSW consists of:

(1) *Item information*, i.e., TIDlist or Bitvector of all 1-itemsets.

(2) High transaction-weighted utilization 2-itemsets maintain in a lexicographical tree.

Table 3-5. The meanings of symbols used in our work

htwu k-itemsets	High transaction-weighted utilization *k*-itemsets
Δ^-	The deleted portion of an ongoing database
D^-	The unchanged portion of an ongoing database
Δ^+	The added portion of an ongoing database

3.2.2.1 Window Initialization Phase

The phase is activated while the number of transactions generated so far in a data stream is less than or equal to a user-specified sliding window size *w*, (i.e. *w* transactions). Initially, the *item information* and transaction utility table are generated by reading transactions. Table 3-4 shows the *item information* in the first two windows. The transaction utility table is shown in Figure 3-2. Once the window is full, we start to build the lexicographical tree.

The procedure to build a lexicographical tree is described as follows. We use **high transaction-weighted utilization 1-itemsets**, denoted as ***htwu 1-itemsets***, to generate candidate 2-itemsets, C_2. As each candidate generates, its transaction-weighted utility is determined immediately by using *item information* and the transaction utility table. We maintain the candidate 2-itemsets whose transaction-weighted utility are above the minimum utility, called **high transaction-weighted utilization 2-itemsets**, denoted as ***htwu 2-itemsets***, in the lexicographical tree.

a, b, c, d and e are *htwu 1-itemsets* in $TransSW_1$. Take item a as an example. Candidate 2-itemsets {ab, ac, ad, ae} are generated from a. The TIDlist for a candidate *k*-itemset is generated by joining the TIDlist of the two *(k-1)*-itemsets that were used to generate the candidate *k*-itemset. The Bitvector for a candidate k-itemset is generated by performing bitwise AND the Bitvector of the two *(k-1)*-itemsets that were used to generate the candidate *k*-itemset. For example, TIDlist(ab) in $TransSW_1$ is {6,8,9} which can be obtained by intersection TIDlist(a) and TIDlist(b), and the Bitvector(ab) in $TransSW_1$ is <000001011> which can be obtained by bitwise AND Bitvector(a) and Bitvector(b). That means itemset ab occur in the sixth, eighth and ninth transactions in $TransSW_1$. Next, *twu(ab)* can be obtained by summation the corresponding transaction utilities. We obtain *twu(ab)*=tu(T6)+tu(T8)+ tu(T9)=48+37+53=138>120 from transaction utility table. Itemsets {ac, ad, ae} are verified in the same way. Finally, there are two *htwu 2-itemsets* from item a. Figure 3-9, shows the tree after generating all *htwu 2-itemsets* from item a. The sub-tree of item b, c, d and e operate the same way as item a. Figure 3-10 shows the tree built in $TransSW_1$.

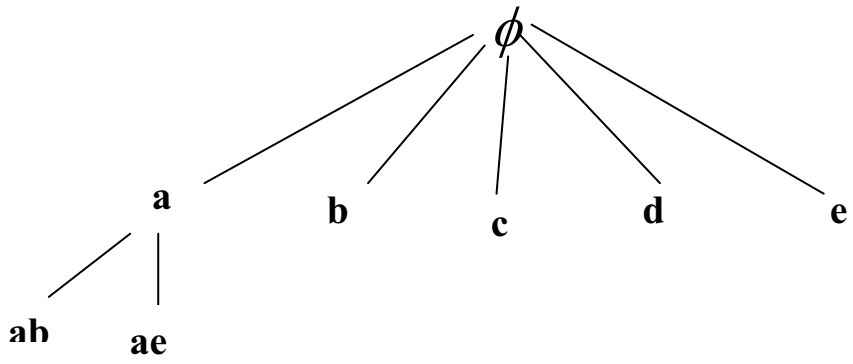


Figure 3-9. The tree after generating all candidate 2-itemsets from item a

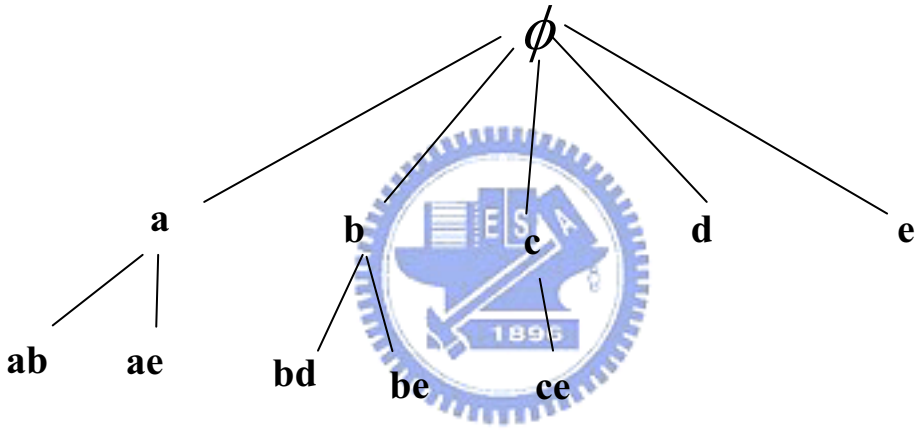


Figure 3-10. The tree built in $TransSW_1$

3.2.2.2 Window Sliding Phase

The window sliding phase is activated while the window is full and new transaction arrives, and window sliding is performed. In this phase, firstly, we update the *item information* and record some extra information. Secondly, we update the tree. We describe these two steps as follows:

1. update *item information*

For removing old transactions, all TIDlist of items are sliding (decrmented by the number

of deleted transactions) and then for adding new transactions, the TIDlist of items in incoming transactions need updating. For removing old transactions, all Bitvector of items perform left shift (shift out the oldest bit) and then for adding new transactions, the Bitvector of items in incoming transactions need updating.

T1 is deleted, so the first transaction is T2 and the last transaction is T10 in $TransSW_2$. Take item c as an example, after deleting T1, TIDlist(c) changes from {1, 5, 8, 9} to {4, 7, 8} and Bitvector(c) changes from<100010011> to <000100110>. Since item c appears in T10, window size, *w,* is added to TIDlist(c) and the latest bit of Bitvector(c) is set to 1. Therefore, TIDlist(c)={4, 7, 8, 9} and Bitvector (c)=<000100111> in $TransSW_2$.

Besides modifying *item information*, we also need to record item in the oldest transaction, denoted as *DeleteItem,* and item in incoming transaction, denoted as *InsertItem*. The oldest transaction, T1, contains item c and e. The incoming transaction, T10, contains item b and c Therefore, *DeleteItem*={c, e} and *InsertItem*={b, c}.

2. update lexicographical tree

After modifying *item information*, MHUI_TransSW begins to modify the tree. Only the sub-trees of the items in the *DeleteItem* or *InsertItem* need to be checked.

This can be decomposed into three parts: The item only in *DeleteItem* is denoted as *OnlyDeleItem. T*he item only in *InsertItem* is denoted as *OnlyInsertItem.* The item not only in *DeleteItem but also in InsertItem* is denoted as *IntersecItem.* Continue the example described above. e is in *OnlyDeleItem*, b is in O*nlyInsertItem*, and c is in *IntersecItem*. Each item in different set performs different operation. The operation in each set is described as follows:

(1) Item in *OnlyDeleIte:* Since the item is only in the oldest transaction, the transaction-weighted utility of its child node may be less than or equal to the previous window. In other words, the child node may be a *htwu 2-itemset* in previous window but is not a *htwu 2-itemset* in current window. We check the child node of this item with *item information* and prune it while its transaction-weighted utility is below the minimum utility. Take item

e as an example. Since there are no potential candidate 2-itemsets from item e carried over from the previous window, no itemsets need to be checked.

(2) **Item in *OnlyInsertItem:*** Since the item is only in the incoming transaction, the transaction-weighted utility of itemsets from it may be larger than or equal to the previous window. In other words, the itemset may be not a *htwu 2-itemset* in previous window but become a *htwu 2-itemset* in current window. We check non-existing itemsets which is from the item with *item information* and insert it while its transaction-weighted utility is greater than the minimum utility. Take item b as an example. *Twu(bc)*=tu(T8)+tu(T9)+ tu(T10)=152>120, so bc is newly inserted into the sub-tree of b. For bd and be, it is not necessary to check because *twu(bd)* and *twu(be)* may only increase but not decrease. Figure 3-11 shows the tree after modifying the sub-trees of items in *OnlyInsertItem.*

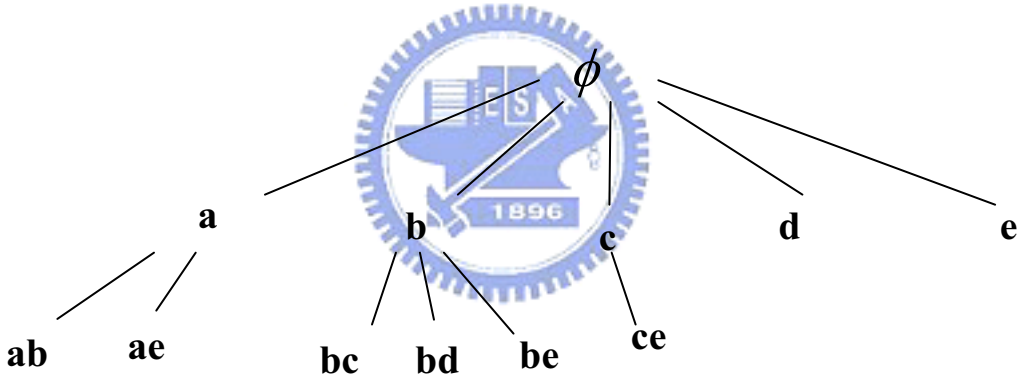


Figure 3-11. The tree after modifing the sub-trees of items in *OnlyInsertItem*

(3) **Item in *IntersecItem:*** Since the item is not only in the oldest transaction but also in incoming transaction, original *htwu 2-itemset* may be not a *htwu 2-itemset* in current window and vice versa. We check the transaction-weighted utility of existing nodes whether it needs to delete. Besides, we check the transaction-weighted utility of non-existing nodes whether it needs to insert. Take item c as an example. cd is a non-existing node in $TransSW_1$, and *twu(cd)*=37<120, and thus it is not newly inserted in $TransSW_2$. ce is an existing node in $TransSW_1$. However, *twu(ce)*=112<120, so it is

deleted in $TransSW_2$. Figure3-12 shows the tree after modifying the sub-trees of items in *IntersecItem.*

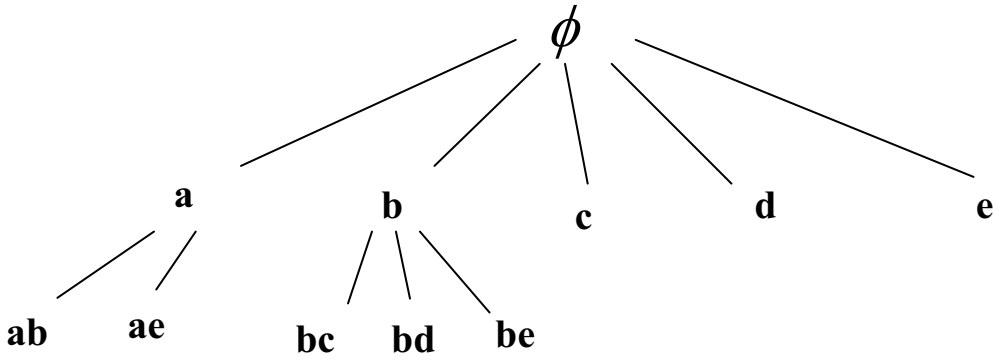


Figure 3-12. The tree after modify the sub-trees of items in *IntersecItem*

3.2.2.3 High Utility Itemsets Generation Phase

In this phase, MHUI_TransSW uses a level-wise method to generate the set of candidate k-itemset , C_k, from the pre-known *htwu (k-1)-itemsets*. Then, we immediately derive the *htwu k-itemsets*, by using *item information* to verify its validity. The candidate-generation-then-testing process stops when no candidates are generated.

Let the minimum utility for nine transactions be 120. An itemset X is a high utility itemset if *u(X)*>=120.There are five *htwu 2-itemsets* generated in $TransSW_1$. Hence only one candidate 3-itemsets {abe} are generated by combining *htwu 2-itemsets***:** ab, ae and be. The TIDlist(abe)={5,7,8}(Bitvector(abe)=<000010110>), so *twu(abe)*=tu(T6)+tu(T8)+tu(T9)=138 >120. Hence, abe is a *htwu 3-itemset*. Because no new candidates are generated, the generation-then-test process stops. After all candidate itemsets are generated, one more scan is needed to find high utility itemsets in $TransSW_1$. Table 3-6 shows the itemsets generated after first and second scan in each window.

Table 3-6. The itemsets generated after first and second scan in each sliding window.

	$TransSW_1$	$TransSW_2$
High transaction-weighted utilization itemsets (itemsets generated after first scan)	a, b, c, d, e ab, ae, bd, be, ce abe	a,b,c,d,e ab, ae, bc, bd, be abe
High Utility Itemsets (itemsets generated after second scan)	b, bd, be	b, bd, be

3.3 The Proposed Method: MHUI_TimeSW

Based on MHUI_TransSW, an efficient method to mine high utility itemsets over a data stream with a time-sensitive sliding window, denoted as MHUI_TimeSW, is proposed in this Section. In section 3.3.1, we describe the time unit in time-sensitive sliding window and the *item information* maintained. We adopt the sliding window model. MHUI_TimeSW consists of three phases, window initialization phase, window sliding phase, and high utility itemsets generation phase, is described from section 3.3.2 to 3.3.4, respectively.

3.3.1 *Item Information* and Time Unit List

A time-sensitive sliding window ($TimeSW$) in the transaction data stream is a window that slides forward for every time unit (TU).The transaction in the time-sensitive sliding window is denoted as T=(TU_{id} , TID, itemset), where TU_{id} is the identifier of the time unit, and TID is the identifier of the transaction. Each time unit consists of variable number, $|TU_i|$, of transactions, and $|TU_i|$ is called the size of the time unit. As the window size is changed as

time advances, MHUI_TimeSW needs to maintain the minimum utility each window required.

Assume the size of time-sensitive sliding window is three and the minimum utility for nine transactions is 120. Figure 3-13 shows the transactions that arrive in the stream in two successive windows, $TimeSW_1 = [TU_1, TU_2, TU_3]$=[T1, T2,…, T6] and $TimeSW_2 = [TU_2, TU_3, TU_4]$=[T3, T4,…, T11]. Table 3-6 shows the transactions contained in each time unit and the size of each time unit. The size of first window is $|TimeSW_1| = |TU_1| + |TU_2| + |TU_3| = 6$ so its minimum utility is 120*(6/9)=80, whereas the size of second window is $|TimeSW_2| = |TU_2| + |TU_3| + |TU_4| = 9$ so its minimum utility is 120. In this example, the deleted portion of an ongoing database, denoted as Δ^-, is TU_1; the added portion of an ongoing database, denoted as Δ^+, is TU_4; the unchanged portion of an ongoing database is TU_2 and TU_3.

	Time Unit	TID	a	b	c	d	e
1	TU1	T1	0	0	26	0	1
2	TU1	T2	0	6	0	1	0
3	TU2	T3	12	0	0	1	0
4	TU2	T4	0	1	0	7	0
5	TU2	T5	0	0	12	0	2
6	TU3	T6	1	4	0	0	1
	TU4	T7	0	10	0	0	1
	TU4	T8	1	1	1	3	1
	TU4	T9	2	1	27	0	2
	TU4	T10	0	6	2	0	0
	TU4	T11	0	3	0	2	0

Relative placement in *TimeSW1*

Relative placement in *TimeSW2*

(a) Transaction table

Item	Profit (per unit)
a	3
b	10
c	1
d	6
e	5

(b) Utility table

Figure 3-13. An example of transaction database and utility table in a time-sensitive sliding window

Table 3-7. The transactions contained in each time unit and the size of each time unit

Time units	Transactions contained	Size of Time Unit
TU_1	TI,T2	2
TU_2	T3,T4,T5	3
TU_3	T6	1
TU_4	T7,T8,T9,T10,T11	5

The representation of *item information* maintained in MHUI_TimeSW is the same as MHUI_TransSW. Table 3-7 shows the *item information* in each window. Noted that the value of TIDlist is less than window size, *w,* and the Bitvector contains *w* bits. Take a as example, a appears in T3, T6 in $TimeSW_1$, and thus TIDlist(a)={3,6} and Bitvector(a)=<001001> in $TimeSW_1$. a appears in T3, T6, T8 and T9 in $TimeSW_2$ and their corresponding placement is first, fourth, sixth and seventh. Hence TIDlist(a)={1,4,6,7}and Bitvector(A)=<100101100> in $TimeSW_2$.

Table 3-8. The *item information* in the first two windows

TIDlist(x) / *Item*	$TimeSW_1$	$TimeSW_2$
a	{3,6}	{1,4,6,7}
b	{2,4,6}	{2,4,5,6,7,8,9}
c	{1,5}	{3,6,7,8}
d	{2,3,4}	{1,2,6,9}
e	{1,5,6}	{3,4,5,6,7}

Bitvector(x) / *Item*	$TimeSW_1$	$TimeSW_2$
a	<001001>	<100101100>
b	<010101>	<010111111>
c	<100010>	<001001110>
d	<011100>	<110001001>
e	<100011>	<001111100>

3.3.2 Window Initialization Phase

The window initialization phase of MHUI_TimeSW is activated while the number of time units generated so far in a transaction data stream is less than or equal to a user-specified time-sensitive sliding window size *w* (i.e. *w* time units). In this phase, first we maintain *item information* and the transaction utility as reading transactions. The *item information* is shown in Table 3-7. The transaction utility table is shown in Figure 3-2.

Next, MHUI_TimeSW builds the lexicographical tree in the same way as MHUI_TransSW. Candidate 2-itemsets are generated by the *htwu 1-itemsets*:{a b, d, e}and we use their corresponding *item information* to verify their validity. Figure 3-14 shows the tree built in $TimeSW_1$. The potential candidate 2-itemsets is {bd}.

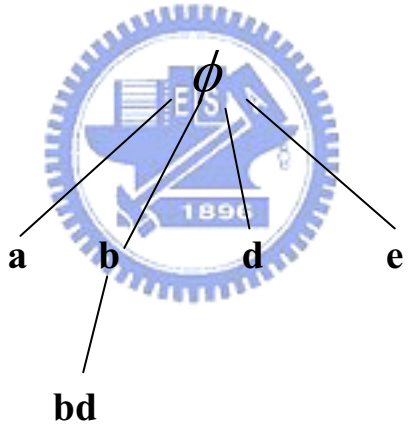


Figure 3-14. The tree built in $TimeSW_1$

3.3.3 Window Sliding Phase

The window sliding phase of MHUI_TimeSW algorithm is activated while the window becomes full. At this time, the oldest time unit is removed from the window, and then a newly one is appended to the time-sensitive sliding window.

In this phase, firstly, we update the *item information* and record some extra information.

Secondly, we update the tree of candidate 2-itemsets. The procedure is the same as MHUI_TransSW except that the minimum utility changes as the window size changes. We describe these two steps as follows.

1. update TIDlist and Bitvector of items

Let $|\Delta^-|$ is the size of deleted portion and $|\Delta^+|$ is the size of added portion. TIDlist is decremented by $|\Delta^-|$ and Bitvector is left shift $|\Delta^-|$ bits for removing the oldest transactions. Next, TIDlist and Bitvector of items in newly time unit need updating. The deleted portion contains two transactions .Take c as an example. TIDlist(c) changes from {1, 5} to {3} and Bitvector(c) changes from <100010> to <001000> after TU_1 is deleted. c is in T8, T9 and T10 which is the sixth, seventh and eighth placement in $TimeSW_2$ respectively. TIDlist(c)={3, 6, 7, 8}and Bitvector(c)=<001001110> after TU_4 is added. Table 3-7, shows the *item information* in each sliding window. Besides updating *item information*, we also need to record *DeleteItem,* and *InsertItem,* the same as MHUI_ TransSW. *DeleteItem* = {b, c, d, e} and *InsertItem*={a, b, c, d, e}.

2. update lexicographical tree

After modification of TIDlist of items, MHUI_TransSW begins to modify the tree. Only the sub-trees of the items in the *DeleteItem* or *InsertItem* need to be checked. This can be decomposed into three parts the same as MHUI_TransSW: *OnlyDeleItem, OnlyInsertItem,* and *IntersecItem.* The operation in each set is the same as MHUI_TransSW mentioned above.

(1) Since there are no elements in *OnlyDeleItem,* we don't do anything.

(2) The item in *OnlyInsertItem:* O*nlyInsertItem is {a}*. Take item a as an example. We check the candidate 2-itemsets {ab, ac, ad, ae} with *item information*. Itemsets {ac, ad} are low transaction-weighted utilization 2-itemsets. On the contrary, itemsets {ab, ae} are *htwu 2-itemsets* and thus are inserted into the tree. Figure 3-15 shows the tree after checking the sub-trees of all items in *OnlyInsertItem.*

(3) The item in *IntersecItem: IntersecItem is {*b, c, d, e}. Take item b as an example.

Itemset {bd} is an existing node. We check whether it becomes a non *htwu 2-itemset*. On the contrary, itemsets {bc, be} are non-existing nodes. We check whether they become *htwu 2-itemsets*. After verification, bd is kept and {bc, be} are inserted. The sub-trees of c, d and e are maintained in the same way. Figure 3-16 shows the tree after checking the sub-trees of all items in *IntersecItem.*

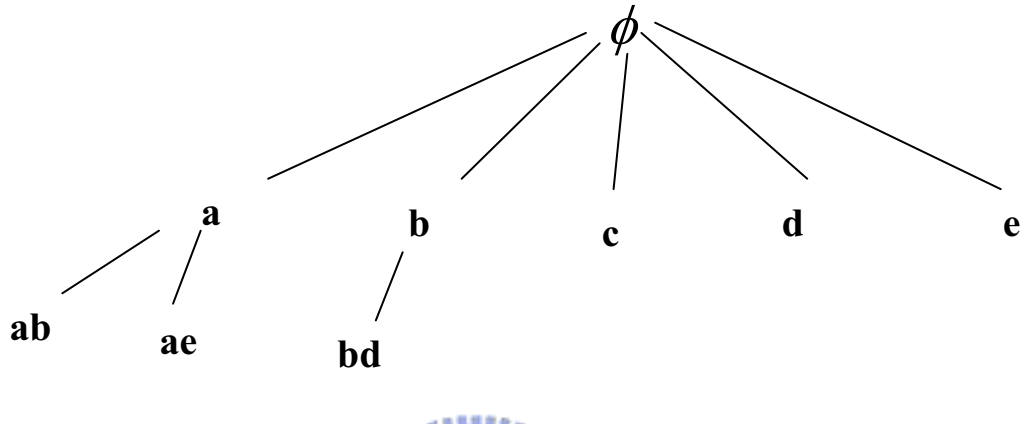


Figure 3-15. After checking the sub-trees of all items in *OnlyInsertItem*

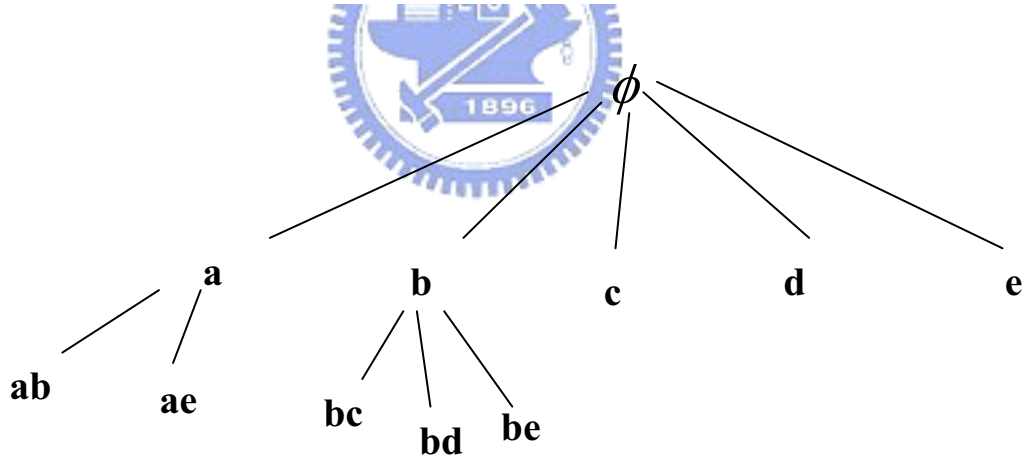


Figure 3-16. After checking the sub-trees of all items in *IntersecItem*

3.3.4 High Utility Itemsets Generation Phase

In the high utility itemsets generation phase, MHUI_TimeSW performs the same as MHUI_TransSW except the minimum utility changes as the window size changes. For example, Table 3-9 shows there are five *htwu 2-itemsets* generated in $TimeSW_2$. Only one candidate 3-itemsets {abe} are generated. The TIDlist(abe)={4,6,7}and Bitvector(abe)=

<000101100>. That is itemset abe occurs in T6, T8 and T9 so *twu(abe)*=tu(T6)+tu(T8)+ tu(T9)=138>120. Itemset abe is a *htwu 3-itemset*. Because no new candidates are generated, the generation-then-test process stops. After all candidate itemsets are generated, one more scan is needed to find the high utility itemsets. Table 3-9 shows the itemsets generated after first and second scan.

Table 3-9. The itemsets generated after first and second scan in each window

	$TimeSW_1$	$TimeSW_2$
High transaction-weighted utilization itemsets (itemsets generated after first scan)	a,b,d,e bd	a,b,c,d,e ab, ae, bc, bd, be abe
High utility itemsets (itemsets generated after second scan)	b, bd	b, bd, be

Chapter 4

Performance Measurement

We perform some experiments to compare our proposed work with THUI-Mine. All the programs are implemented in C++ STL and compiled with Visual C++.NET compiler. All the programs are performed on AMD Athlon(tm) 64 Processor 3000+ 1.8GHz with 1GB memory and running on Windows XP system.

All testing data was generated by the synthetic data generator provided by Agrawal et al in [2]. However, the IBM generator only generates the quantity of 0 or 1 for each item in a transaction. In order to adapt the databases into the scenario of utility mining, the quantity of each item and the utility of each item is randomly generated. The meaning of symbols is shown in Table 4-1.

Table 4-1. Meanings of symbols used

|W|	Window size
|P|	Partition size
ut	Minimum utility threshold
Q_{ip}	The quantity of each item in each transaction
U_{ip}	Utility of each item
Δ^+	The added portion of ongoing database
Δ^-	The deleted portion of ongoing database
Item_freq	The frequent of item, i.e., the average number of TIDlist of all items. That is the average number of transactions each item contained.

In our programs, we randomly generate Q_{ip}, ranging from 1 to 5. U_{ip}, stored in utility table, is also synthetically created by assigning a utility value to each item randomly, ranging from 1 to 1000. Observed from real world databases that most items are in the low profit range, the utility value generated using a log normal distribution, as is similar to the model used in THUI-Mine. We use several sets of synthetic databases from IBM generator. Table 4-2 shows the names and parameter settings for each data set. Our testing metric includes the number of candidate itemsets generated, execution time and memory consumed.

Table 4-2. The names and parameter settings for each data set.

	Average items per transaction (T)	Average length of maximal pattern(I)	Number of transactions(D)	Number of items
T5I4D100K	5	4	100K	1000
T10I6D100K	10	6	100K	1000
T15I10D100K	15	10	100K	1000
T20I15D100K	20	15	100K	1000

4.1 Experiments of MHUI_TransSW Method

In this section, we compare the mining results of MHUI_TransSW and THUI-Mine using the same dataset T5I4D100K. The number of item types is fixed to 1,000. The sliding window is fixed to 5,000 transactions and the partition size is fixed to 1 transaction.

4.1.1 Different Minimum Utility Threshold

In this section, we test the execution time, memory usage and number of candidate itemsets

generated under different minimum utility thresholds where ut is changed from 0.9% to 6.0%.

1. **Execution time:** Figure 4-1 shows the result of execution time. Observe that MHUI_TransSW runs efficiently faster than THUI-Mine in a transaction-sensitive sliding window.

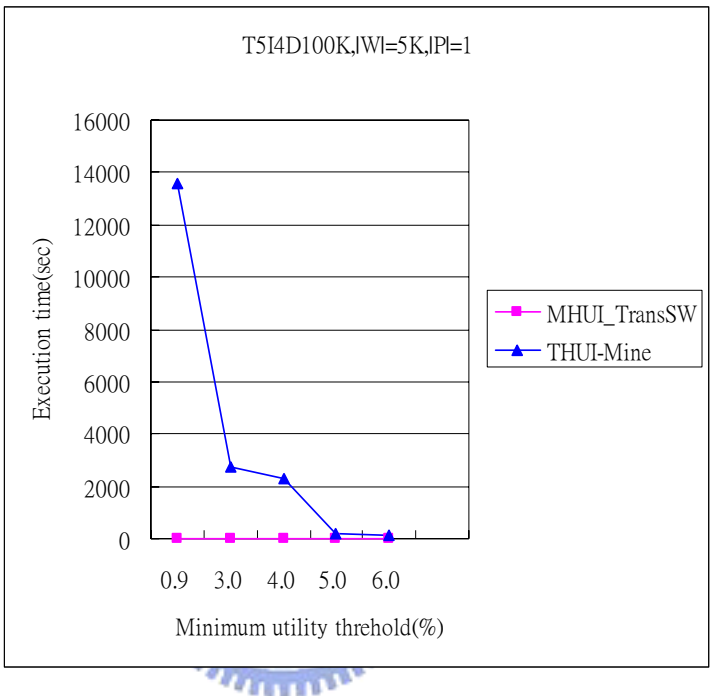


Figure 4-1. The execution time of MHUI_TransSW and THUI-Mine under different minimum utility thresholds

2. **Memory usage:** Figure 4-2 shows the result of memory usage .Observe that the memory used in MHUI_TransSW is almost the same (this is obvious from that the number of candidate itemsets generated increases little under these different utility thresholds). However, the memory used in THUI-Mine increases dramatically as the minimum utility threshold decreases.

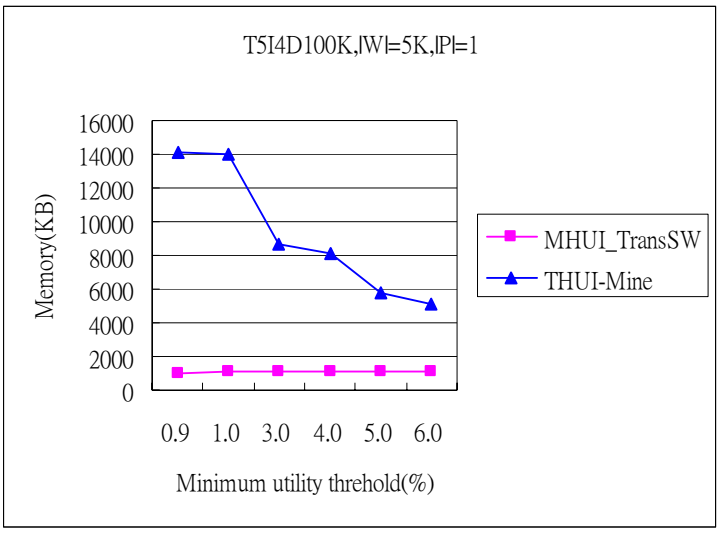


Figure 4-2. The memory usage of MHUI_TransSW and THUI-Mine under different minimum utility thresholds

3. **The number of candidates generated after 1st scan:** Observe that the smaller the minimum utility threshold is the larger the number of the candidates THUI-Mine generates. Table 4-3 shows the number of candidates generated.

Table 4-3. The number of candidates generated by MHUI_TransSW and THUI-Mine with different minimum utility thresholds

Minimum utility threshold (%)	THUI-Mine	MHUI_TransSW
0.9	218521	15
1.0	217162	8
2.0	138062	0
3.0	98203	0
4.0	87019	0
5.0	20885	0

From previous experiments, we verify that MHUI_TransSW generates less candidate itemsets so that MHUI_TransSW runs significant faster and consumes less memory when using transaction-sensitive sliding window.

4.2 Experiments of MHUI_TimeSW Method

In this section, we compare the mining results of MHUI_TimeSW and THUI-Mine using the same dataset T10I6D100K. The number of item types is fixed to 1,000. The sliding window is fixed to 30,000 transactions. The partition size is fixed to 10,000 transactions. Without loss of generality, we set $|P| = |\Delta^+| = |\Delta^-| = 10{,}000$ for simplicity, where |P| denotes the partition size, $|\Delta^+|$ denotes the size of added portion, and $|\Delta^-|$ denotes the size of deleted portion.

4.2.1 Different Minimum Utility Threshold

In this section, we test the execution time, memory usage and the number of candidate itemsets generated under different minimum utility thresholds where ut is changed from 0.3% to 1.0%.

1. **Execution time:** Figure 4-3 shows the result of execution time. As minimum utility threshold is larger than 0.5%, MHUI_TimeSW in average is two times faster than THUI-Mine. However, as the minimum utility threshold is less than 0.5%, the performance difference becomes prominent in that MHUI_TimeSW significantly outperforms THUI-Mine.

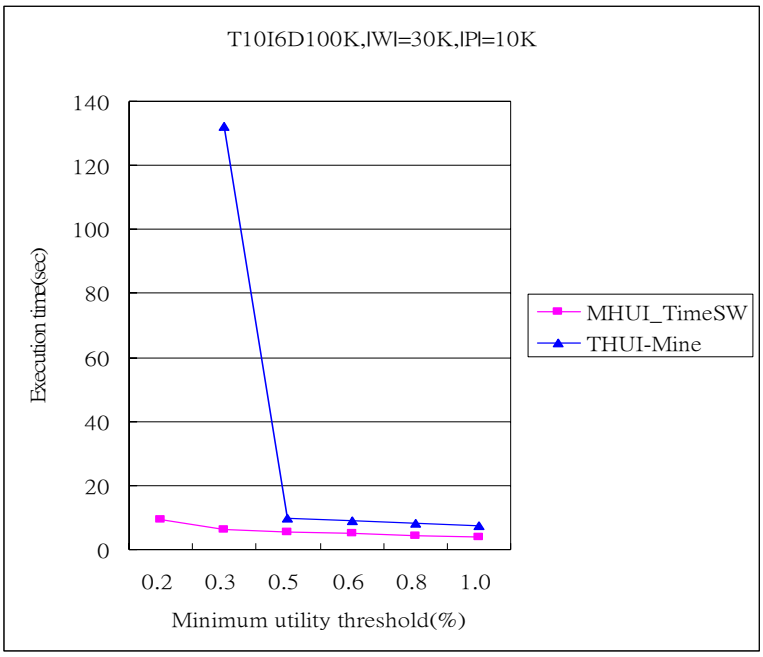


Figure 4-3. The execution time of MHUI_TimeSW and THUI-Mine with different minimum utility thresholds

2. **Memory usage:** Figure 4-4 shows the result of memory usage. Observe that the memory usage of MHUI_TimeSW is almost the same (this is obvious from that the number of candidate itemsets generated increases little under these different utility thresholds). However, the memory used in THUI-Mine is getting larger as the minimum utility threshold decreases.

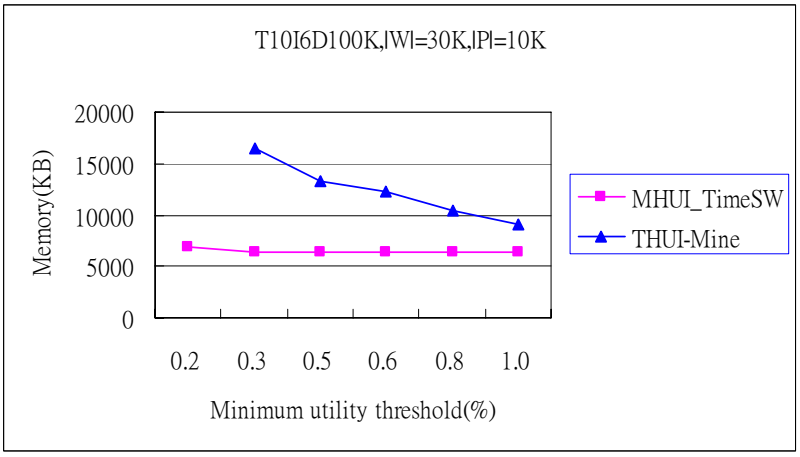


Figure 4-4. The memory usage of MHUI_TimeSW and THUI-Mine with different minimum utility thresholds

3. **The number of candidates generated after 1st scan:** Observe that the smaller the minimum utility threshold is the larger the number of the candidates THUI-Mine generates. Table 4-4 shows the number of candidates generated by MHUI_ TimeSW and THUI-Mine with different minimum utility thresholds.

Table 4-4. The number of candidates generated of MHUI_TimeSW and THUI-Mine with different minimum utility thresholds

Minimum utility threshold (%)	THUI-Mine	MHUI_TimeSW
0.3	10788	672
0.5	102	38
0.6	18	7
0.8	1	1
1.0	0	0

We conclude that the execution time, memory usage and the number of candidate itemsets of THUI-Mine will increase significantly as minimum utility threshold decreases.

4.2.2. Different Partition Size

In this section, we compare the mining results of MHUI_TimeSW and THUI-Mine using the same dataset as Section 4.2.1 except that ut is fixed and partition size is variable in this Section. The minimum utility threshold is fixed to 0.5%.

We test the execution time, memory usage and the number of candidate itemsets generated under different partition size where |P| is changed from 1 to 15,000.

1. **Execution time:** Figure 4-5 shows the result of execution time. When partition size is larger than 5K, MHUI_TimeSW in average is two to three times faster than THUI-Mine.

As the partition size is less than 5K, the performance difference becomes prominent in that MHUI_TimeSW significantly outperforms THUI-Mine. The reason is that as the partition size decreases, the number of false candidates becomes larger. Hence, it will need much more time to process.

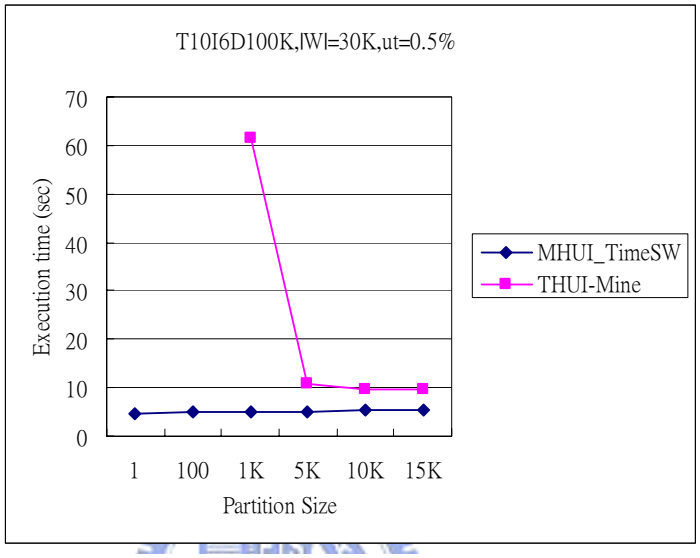


Figure 4-5. The execution time of MHUI_TimeSW and THUI-Mine with different partition sizes

2. **Memory usage:** Figure 4-6 shows the result of memory usage. Observe that as partition size is larger than 1K, the memory required in MHUI_TimeSW in average is two to three times larger than THUI-Mine. As partition size is less than 1K, the memory required in THUI-Mine increases dramatically. The mainly memory usage depends on two factors, one is the extra resource each algorithm maintained; the other is the candidate itemsets generated. For the former, THUI-Mine maintains $TUP_K(I)$ which varies with the number of candidate 2-itemsets, whereas MHUI_TimeSW maintains *item information* which is fixed under the same dataset and window size. For the latter, the number of candidate itemsets THUI-Mine generated is much more than MHUI_TimeSW, especially when partition size decreases (see Table 4-5).

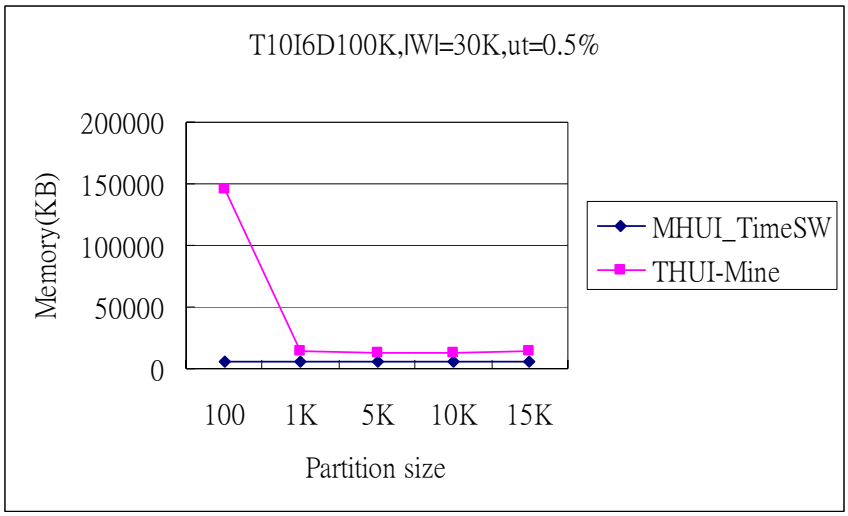


Figure 4-6. The memory usage of MHUI_TimeSW and THUI-Mine with different partition sizes

3. **The number of candidate itemsets generated after 1st scan:** Table 4-5 shows the result of candidates generated. MHUI_TimeSW is not a partition-based method, so change of partition size doesn't affect the number of candidates generated. On the contrary, THUI-Mine is deeply influenced. Observe that as the partition size is less than 1K, the number of candidates THUI-Mine generated increase significantly.

We conclude that the execution time, memory usage and the number of candidate itemsets of THUI-Mine will increase dramatically as partition size decreases.

Table 4-5. The number of candidates generated of MHUI_TimeSW and THUI-Mine with different partition sizes.

Partition size	THUI-Mine	MHUI_TransSW
1K	8269	37
5K	181	37
10K	102	38
15K	71	39

4.3 The Performance between TIDlist and Bitvector

We can use TIDlist or Bitvector, to store *item information* as mentioned in Section 3.2.1. In Section 4.1 and 4.2, our method chooses TIDlist representation, because the performance of TIDlist is better than Bitvector under those datasets. In this section, we compare our proposed method using TIDlist and Bitvector, denoted as **MHUI(TID)** and **MHUI(BIT)** respectively, with THUI-Mine. We show the execution time of different methods with different minimum utility thresholds.

The first experiment, we use dataset T10I6D100K to compare the performance of these three methods. The parameter setting is the same as in Section 4.2.1. Figure 4-7 shows the execution time of these three methods under different minimum utility thresholds.

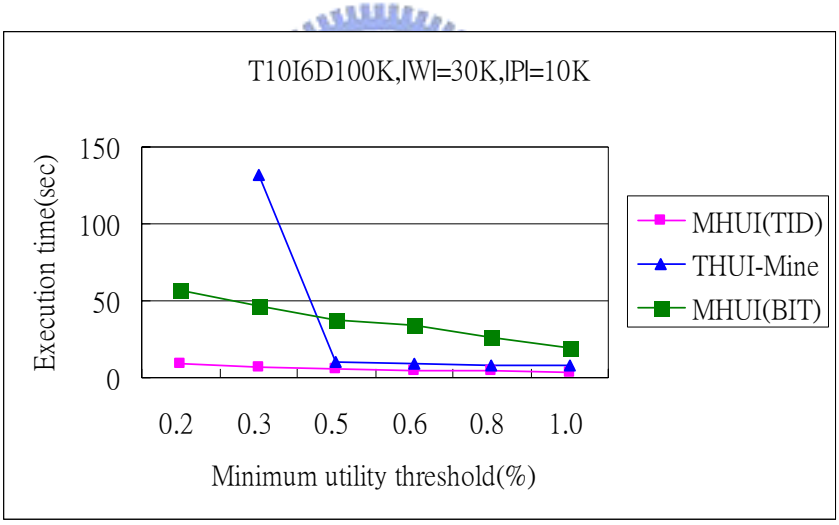


Figure 4-7. The execution time of these three methods under different minimum utility thresholds

Observe that MHUI(TID) always performs better than THUI-Mine, especially when utility threshold is small. MHUI(BIT) runs a little slowly than THUI-Mine when utility threshold is larger than 0.5%, however, MHUI(BIT) runs significantly faster than THUI-Mine when utility threshold is less than 0.5%.

Next, we use three datasets, T15I10D100K, T20I15D100K and T30I20D100K, to compare the performance of these three methods. The window size is fixed to 30,000. The partition size is fixed to 10,000. The minimum utility threshold is fixed to 1%. Figure 4-8 shows the execution time of these three methods. THUI-Mine only runs successfully in dataset T15I10D100K. However, THUI-Mine cannot draw in the picture since that it needs much more time corresponding than the MHUI(TID) and MHUI(BIT). Observe that in these three datasets, the execution time needed is MHUI(TID) << MHUI(BIT) << THUI-Mine. Although MHUI(BIT) does not completely win THUI-Mine in dataset T10I6D100K, but it completely beats THUI-Mine in larger dataset, such as T15I10D100K, T20I15D100 and T30I20D100K. In other words, MHUI(BIT) maybe runs a little slower than THUI-Mine in a smaller dataset, it maybe runs significantly faster than THUI-Mine in a larger dataset.

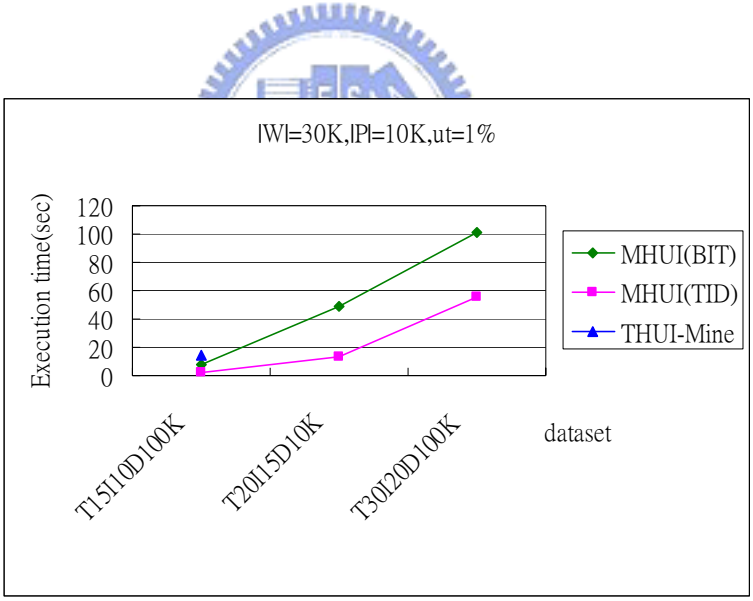


Figure 4-8. The execution time of these three methods under different datasets

Table 4-6 shows the *Item_freq*, the average number of transactions containing each item. Observe that *Item_freq* increases a little as the dataset become larger. The ratio of *Item_freq* to *window size* is from 480/30,000=1.6% to 750/30,000=2.5% in T15I10D100K to T30I20D100K respectively. Since the ratio is apparently small, we can obtain that the

performance of MHUI(TID) is better that MHUI(BIT). Figure 4-7 and Figure 4-8 verified the conclusion.

Table 4-6. The *Item_freq* in each dataset

Dataset	*Item_freq*
T15I10D100K	480
T20I15D100K	550
T30I20D100K	750

4.4. The Stability of Our Proposed Work

In this experiment, we examine the two primary factors, execution time and memory usage, to discovery high utility itemsets in a data stream environment. We use there datasets, T10I6D100K, T15I20D100K and T20I15D100K, and change one parameter at a time to prove the stability of our proposed work. Each is described as follows:

1. **Different Minimum Utility Thresholds:** In the first experiment, the window size is fixed to 30,000 and the partition size is fixed to 10,000. We test the execution time and memory usage under different minimum utility thresholds where ut is changed from 0.5% to 1.0%. Figure 4-9 and Figure 4-10 show the execution time and memory usage, respectively.

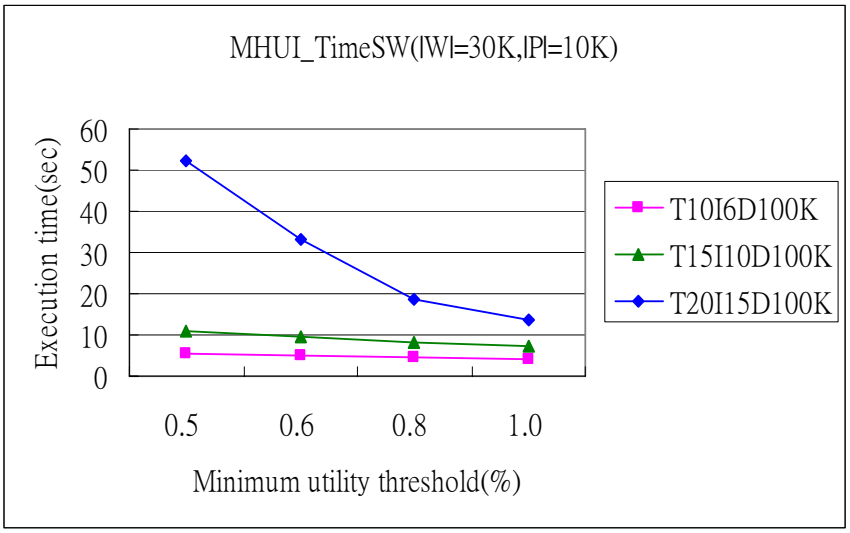


Figure 4-9. The execution time of MHUI_TimeSW under different minimum utility thresholds

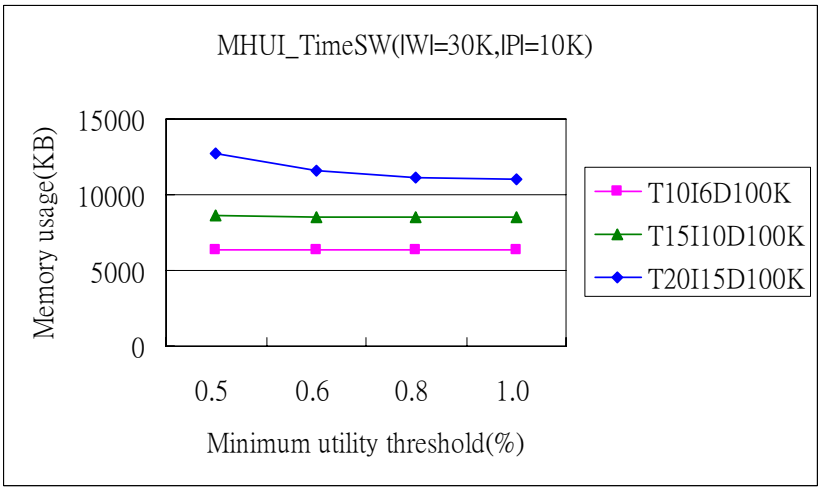


Figure 4-10. The memory usage of MHUI_TimeSW under different minimum utility thresholds

2. **Different Partition Size:** In the second experiment, the window size is fixed to 30,000 and the minimum utility threshold is fixed to 0.5%. We test the execution time and memory usage under different partition sizes where |P| is changed from 100 to 15,000. Figure 4-11 and Figure 4-12 show the execution time and memory usage respectively.

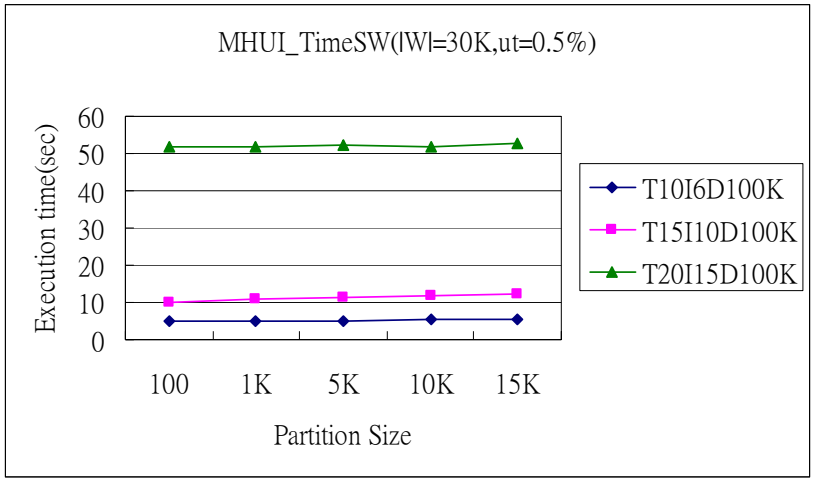


Figure 4-11. The execution time of MHUI_TimeSW under different partition sizes

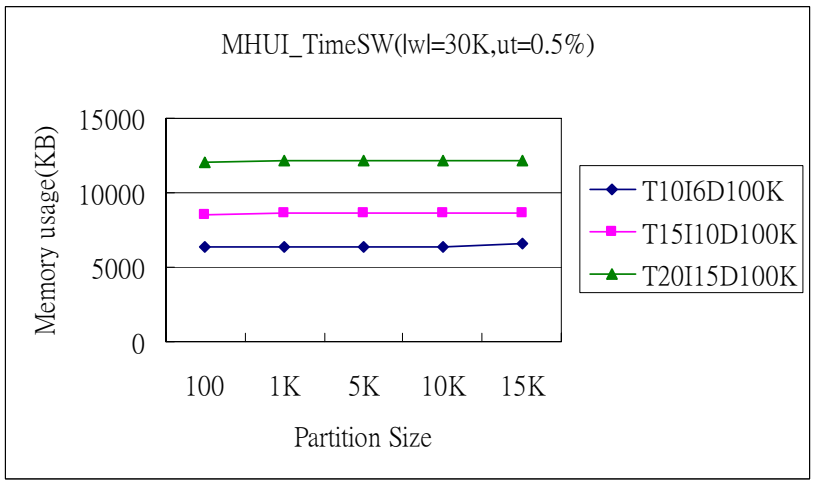


Figure 4-12. The memory usage of MHUI_TimeSW under different partition sizes

3. **Different Window Size:** In the third experiment, the partition size is fixed to 10,000 and the minimum utility threshold is fixed to 1.0%. We test the execution time and memory usage under different window sizes where |W| is changed from to 20,000 to 60,000. Figure 4-13 and Figure 4-14 show the execution time and memory usage, respectively.

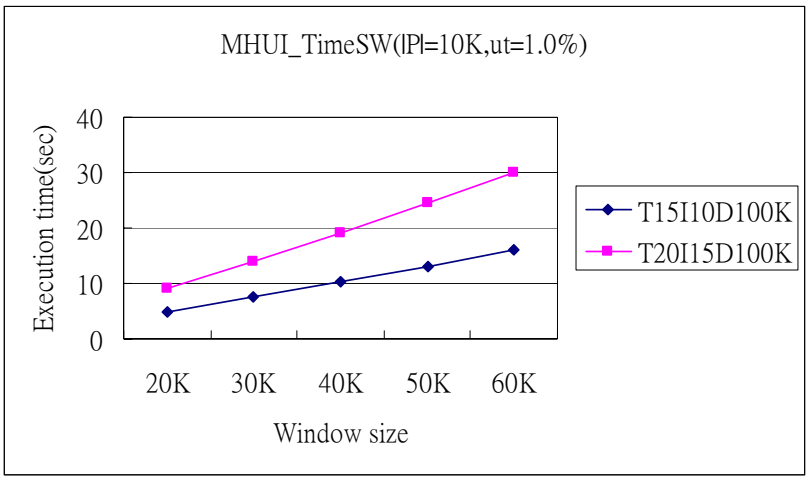


Figure 4-13. The execution time of MHUI_TimeSW under different window sizes

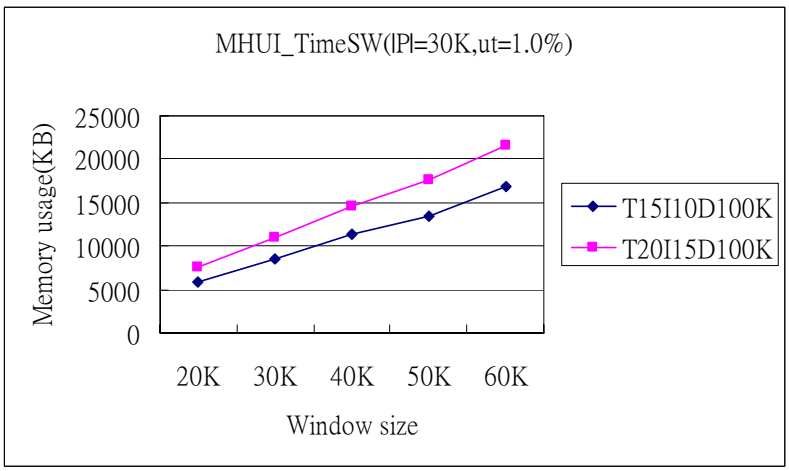


Figure 4-14. The memory usage of MHUI_TimeSW under different window sizes

Observe that the execution time and memory usage goes smoothly as time advances no matter we change what kind of parameters. This result indicates that our proposed work is stable and fit for all kinds of datasets.

Chapter 5

Conclusion and future work

5.1 Conclusion of Our Proposed Work

Due to the limitation of data streams and the complexity of computing utility itemsets, mining of high utility itemsets in a data stream is more complicated than in static database. In this thesis we propose two methods: MHUI_TransSW and MHUI_TimeSW to mine the high utility itemsets with the transaction-sensitive and time-sensitive sliding window respectively.

THUI-Mine is the first algorithm proposed to find high utility itemsets in a data stream. THUI-Mine uses partition-based to find the candidate itemsets, and thus it generates too many candidate itemsets and need more time and memory to find the high utility itemsets. The goal of our work is to improve on mining high utility itemsets in execution time, memory usage and the number of candidates generated. Our work is based on the transaction-weighted downward closure property and utilizes efficient *item information*, i.e, TIDlist or Bitvector, and additionally builds a lexicographical tree to maintain the candidate 2-itemsets.

Experiments show that execution time and memory usage of MHUI_TransSW significantly outperforms THUI-Mine in a transaction-sensitive sliding window. Next, we extend MHUI_TransSW to MHUI_TimeSW. Experiments validate the efficiency. For smaller datasets and smaller item_freq/window, MHUI_TimeSW using TIDlist as *item information* runs averagely two times faster than THUI-Mine, and the margin grows as the minimum utility threshold decreases or the partition size decreases. MHUI_TimeSW using Bitvector as *item information* runs a little slower than THUI-Mine, whereas it runs significantly faster as the partition size decreases or the minimum utility threshold decreases. For larger datasets and smaller item_freq/window size, no matter MHUI_TimeSW uses TIDlist or Bitvector as the *item information*, it runs significantly faster and consumes less memory space than THUI-Mine

especially when the partition size is small or the minimum utility threshold is small.

5.2 Future Work

Our work uses the transaction-weighted utilization property to filter out the candidate itemsets, so it needs two scans, the first scan to find high transaction-weighted utilization itemsets and the second scan to find the high utility itemsets. However, one of the characteristics of data streams is expiration, which means data can be read only once. Because of the complicated calculation of utility itemsets it is challenging to mine high utility itemsets on data streams in one-pass scan.

Bibliography

1. R. Agrawal, T. Imielinski, A. Swami, Mining associations rules between sets of items in large Databases, In Proc. of ACM SIGMOD Intel. Conf. on Management of Data, pp. 207-216, 1993.
2. R. Agrawal and R. Srikant, Fast Algorithms for Mining Association Rules in Large Database, In Proc. of the 20th Intel. Conf. on Very Large Databases (VLDB), pp. 487-499, 1994.
3. A. Savasere, E. Omiecinski and S. Navathe, An Efficient Algorithm for Mining Association Rules in Large Database, In Proc. of the 21th Intel. Conf. on Very Large Databases (VLDB), pp. 432-444, 1995.
4. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo, Fast Discovery of Associations Rules, *Advances in Knowledge Discovery and Data Mining*, pp. 307-328. AAAI/MIT Press, 1996.
5. J. S. Park, M. S. Chen, P. S. Yu, Using a Hash-Based Method with Transaction Trimming for Mining Association Rules, IEEE Trans. on Knowledge and Data Engineering, 9(5): pp. 813-825, 1997.
6. J. Han, J. Pei, and Y. Yin, Mining Frequent Patterns without Candidate Generation, In Proc. of ACM SIGMOD Intel. Conf. on Management of Data, pp. 1-12, 2000.
7. C. H. Lee, C. R. Lin and M. S. Chen, Sliding-Window Filtering: An Efficient Algorithm for Incremental Mining, In Proc. of the ACM 10th Intel. Conf. on Information and Knowledge Management (CIKM), pp. 263-270, 2001.
8. Y. Zhu, D. Shasha**,** StatStream: Statistical Monitoring of Thousands of Data Stream in Real Time, In Proc. of the 28th Intel. Conf. on Very Large Databases (VLDB), pp. 358-369, 2002.
9. G. Manku and R. Motwani, Approximate Frequency Counts over Data Streams, In Proc.

of the 28th Intel. Conf. on Very Large Databases (VLDB), pp.346-357, 2002.

10. C. Jin, W. Qian, C. Sha, J. Yu, A. Zhou, Dynamically Maintaining Frequent Items over a Data Stream, In Proc. of the ACM 12th Intel. Conf. on Information and Knowledge Management (CIKM), pp. 287 - 294, 2003.
11. L. Golab and M. T. Ozsu, Issues in Data Stream Management, In ACM SIGMOD Record, 32(2): pp. 5-14, 2003.
12. R. Chan, Q. Yang, Y. D. Shen, Mining High utility Itemsets, In Proc. of the 3rd IEEE Intel. Conf. on Data Mining (ICDM), 2003
13. J. H. Chang and W. S. Lee, Finding Recent Frequent Itemsets Adaptively over online Data Streams, In Proc. of Intel. Conf. on Knowledge Discovery and Data Mining (SIGKDD), pp.487-492, 2003.
14. J. Chang and W. Lee, "A Sliding Window Method for Finding Recently Frequent Itemsets over online Data Streams", Journal of Information Science and Engineering, 20(4): pp. 753 - 762, 2004.
15. H. F. Li, S. Y. Lee and M. K. Shan, DSM-FI: An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams, In 1st Intel. Workshop on Knowledge Discovery in Data Streams, 2004.
16. Y. Chi, H. Wang, P. S. Yu, R. Muntz, Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window, In Proc. IEEE Intel. Conf. on Data Mining (ICDM), pp. 59-66, 2004.
17. H. Yao, H. J. Hamilton, and C. J. Butz, A Foundational Approach to Mining Itemset Utilities from Databases, In Proc. of 4th SIAM Intel. Conf. on Data Mining (SDM), 2004.
18. C. H. Lin, D. Y. Chiu, Y. H. Wu, A. L. P. Chen, Mining Frequent Itemsets from Data Streams with A Time-Sensitive Sliding Window, In Proc. of SIAM Conf. on Data Mining (SDM), 2005.

19. Y. Liu, W. Liao, and A. Choudhary, A Fast High Utility Itemsets Mining Algorithm, In Proc. of the ACM Intel. Conf. on Utility-Based Data Mining Workshop (UBDM), 2005.
20. V. S. Tseng, C. J. Chu, and T. Liang, Efficient Mining of Temporal High Utility Itemsets from Data Streams, In Proc. of the ACM Intel. Conf. on Utility-Based Data Mining Workshop (UBDM), 2006.
21. H. Yao, H. Hamilton and L. Geng, A Unified Framework for Utilty-Based Measures for Mining Itemsets, In Proc. of the ACM Intel. Conf. on Utility-Based Data Mining Workshop (UBDM), pp. 28-37, 2006.